Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

INFORMATION ENGINEERING DEPARTMENT

COMPUTER ENGINEERING - AI & ROBOTICS

Deep Learning 2023/2024

Francesco Crisci 2076739

Pattern and ML Recalls

Data and Pattern

Data is a fundamental ingredient of machine learning, where the behavvior of the algorithms is not pre-programmed but learned from the data itself. We can have different kind of patterns:

- **Numeric:** values associated with measurable characteristics. Typically continuous and subject to order. Naturally represented as numerical vectors in the multidimensional space.
- **Categorical:** values associated with qualitative characteristics and the presence or absence of a feature. Not "semantically" mappable into numerical values.
- **Sequences:** sequential patterns with spatial or temporal relationships. Often variable in length. The position in the sequence and the relationships with predecessors and successors are important.

Local Binary Pattern (LBP)

TThe basic idea is that two-dimensional textures can be desccribed by two complementary measures: local spatial patterns and gray scale contrast. It was then extended to use neighborhoods of different size. Using a circular neighborhood and bilinearly interpolating values at non-integer pixel coordinates allow any radius and number of pixels in the neighborhood. The gray scale variance of the local neighborhood can be used as the complementary contrast measure. The notation (P,R) will be used for pixel neighborhoods which means P sampling on a circul of radius R. The value of the LBP code of a pixel (x_c, y_c) is given by:

$$LBP_{P,R} = \sum_{p=o}^{P-1} s(g_p - g_c)2^p \qquad s(x) = \begin{cases} 1 \text{ if } x \geq 0 \\ 0 \text{ otherwise} \end{cases}$$

with $x_p = x_c + Rcos(\frac{2\pi P}{P})$ and $y_p = y_c + Rsin(\frac{2\pi P}{P})$. The descriptors just seen, if calculated on the entire image, do not retain spatial information; to enrich the descriptors with local information it is possible to divide the image into regions calculating the descriptors for each region.

Dissimilarity Space

The original definition of dissimilarity space is the following:
The dissimilarity space is a vector space in which the dimensions are defined by dissimilarity vectors measuring pariwise dissimilarities between exaples and individual objects from the so-called representation set R. Hence, a dissimilarity representation D(X,R) is addressed as a data-dependent mapping $D(\bullet, R) : X \to \mathbb{R}^n$ from an initial set of objects X to a dissimilarity space,

equipped with the traditional inner product and Euclidean metric. The representation set can be chosen as the complete traiining set T, a set of carefully selected or constructed prtotypes.
Assume n objects, $O = o_1, ..., o_n$ and an $n \times n$ dissimilarity matrix D:=D(O,O). $D_{ij} = d(o_i, o_j)$ is the dissimilarity between the sensor inputs for objects o_i and o_j. $D(O, o_k) := [d(o_1, o_k), ..., d(o_n, o_k)]^T$ is a feature defined by pairwise dissimilarities to o_k. $x_i := D(o_i, O) = [d(o_i, o_1), ..., d(o_i, o_n)]^T$ is a dissimilarity-based representation for o_i. X:=D, defines an n-dimensional vector space in which the dimension k is defined by $D(O, o_k)$ and the vector x_i represents o_i.
In pracctice, the dissimilarity space methods make possible to represent complex patterns in vector format even without the dissimilarity sppace being a metric.

Metric

A metric space is an ordered pair (M,d) where M is a set and d is a metric on M, i.e., a function $d : M \times M \to \mathbb{R}$ such that for any $x, y, z \in M$ the following holds: $d(x, y) = \Leftrightarrow x = y,\ d(x, y) = d(u, x),\ d(x, z) \leq d(x, y) + d(y, z),\ d(x, y) \geq 0$.

Classification

Is the task to assign a class to a pattern. It is necessary to learn a functio ncapabble of performinh the mapping from pattern space to class pattern. In case we have only two classes we use the term binary classification, otherwie we use multi-class classification.

Regression

The Regression task assiigns a continuus value to a pattern. Solving a regression problem corresponds to learn an approximant function of the pairs input-output. The regression problem requires the prediction of a quantity. It can have real or discrete input variables. A problem with multiple input variables is called a multivariate regression problem. If the input variable are ordered by time , then the problem is called time series foreccasting problem.

Clustering

It is the task to group together similar objects and to put in different groups different objects. The problem is that classes are not known and the patterns are not labeled. Often, not even the number of clusters is known at priori. The clusters identified in learning can then be used as classes. In computer vision, image segmentation is the process of partitioning a digital image into multiple segments. The goal of segmentation is to simplify and/or change the representation of an image into something that is more meaningful and easier to analyze. Image segmentation is used to locate objets and boundaries in images. More precisely, it assign a label to every pixel in an image such that pixels with the same label share certain characteristics.

Dimensionality Reduction

Dimensionality reduction is the transformation of data from a high-dimensional space into a low-dimensional space; so that the low-dimensional representation retains some meaningful properties of the original data. It is a mapping from $\mathbb{R}^d \to \mathbb{R}^k$, where k is the number of feature used to describe a pattern instead of using the original d features. We have different dimentionality reduction methods:

- Only keep the most important features, e.g., Backward elimination, Forward selection or Random forests;
- Find a combination of new features, which can be split in:
  - Linear methods, e.g. PCA, FA, LDA or Truncated SVD;
  - Non-liner methods (Manifold learning), e.g., Kernel PCA, t-SNE, MDS or Isomap.

Feature learning

The succes of many ML applications depends from the effecctiveness of the representation of the patterns in terms of features. The definition of ad-hoc features for the different applications is called feature engineering. Most of the deep-learning techniques operate using raw data as input and then they extracct automatically te features necessary to solve the problem off interest.

Learning Approaches

We have 3 kinds of learning approaches:

- **Supervised:** pattern classes are knwon, the training set is labeled. Typical situation in classification, regression and some dimension reduction techniques.
- **Unsupervised:** the classes of the patterns used for training are unknown, the training set is not labeled. Typical situation in clustering and in most of the dimensionality reduction techniques.
- **Semisupervised:** the training set is partially labeled, unlabeled patterns can help to optimize the classification rule.

Supervised learning

Given pre-classified examples, $Tr = \{(x^{(i)}, t^{(i)})\}$, learn a general description which captures the information content of the examples. The model will learn a function h such thta $h(x^{(i)}) \approx f(x^{(i)})$ for all the examples in Tr. It should be possible to use this description in a predictive way.

Unsupervised learning

given a set of examples $Tr = \{x^{(i)}\}$, discover regularities and/or patterns. There is no expert or teacher to give supervision.

One/few shot learning

It is an object categorization problem, found mostly in computer vision. It aims to classify objects from one, or only few, samples. The key motivation for solvving one-shot learning is that systems, like humans, can use knwoledge about object categories to classify new objects.

Zero shot learning

Zerp-shot learning is an approach in machine learning that takes inspiration from the facct that we can recognize things we have never seen. In a zero-shot learning approach we have the data in the following manner: seen classes (classes with labels available for training), unseen classes (classes that occur only in the test set or during inference. Not present during training), and auxiliary information (information abouth both seen and unseen class lables during trianing time).

Sequential Learning

Models that are able to either ingest sequences of inputs or to emit sequences of outputs, or sometimes doing both. Specifically sequence to sequence learning considers problems where input and output are both varaible-length sequences, such as machine translation and transcribing text ffrom spoke speach.

Offline learning

In case of both supervised or unsupervised learning we grab a big pile of data upfront, then set our pattern reccognition machines in motion without ever interacting with the environment again. Because all of the learning takes place after the algorithm is disconnected from the environment, this is sometimes called offline learning.

Reinforcement learning

The usual choice in case you wanna develop an agent that interacts with an environment and takes actions. An agent interacts with an environment over a series of time steps. At each step, the agent receives some observation from the environment and must choose an action that is subsequentely transmitted back to the environment via some mechanism. Finally the agent recceives a reqord from the environment. More formally we have:

- **Agent** which may be in state s and execute an action
- **Environment e** which in response to action a in the state s returns the next action and a reqard r, which can be positive, negative or neutral.

The goal of the agent is to maximize a function of the rewards, like $\sum_{t=0}^{\infty} \gamma^t r_{t+1}, with 0 \leq \gamma < 1$.

Continual learning

It is build on the idea of learning continuously and adaptively about the external world and enabling the autonomous incremental development of ever more complex skills and knowledge.

Different kinds of learning

We have two main families:

- **Batch:** the training is done only once on a training set. Once the training is finished, the system switches to workin modality and it is unable to learn any further.
- **Incremental:** following initial training there are further training sessions. We have batsh sequences and unsupervised tuning. We might have catastrohpic forget, but it follows the natural concept of continuous learning. It allows coexistence of supervised and unsupervised approaches.

Parameters optimization

The behavior of a machine learning algorithm is regulated by a set of parameters. We want to determine the optimal value of these parameters. Giving a training set and a set of parameters, the objective function can indicate the optimality of the solution

$$\Theta^* = \text{argmaax}_\Theta f(Train, \Theta)$$

In case we want to minimize the training error:

$$\Theta^* = \text{argmin}_\Theta f(Train, \Theta)$$

Hyperparameters

Many algorithms require to define the value of the so-called hyperparameters, like number of neurons in a NN, number of neighbors in a k-NN classifier and so on. Usually, you could proceed with a two-level approach in which for each reasonable value of the hyperparameters is performed a learning phase, and at the end of the procedure the hyperparameters that performed better are chosen.

Performance indicators

We can define two metrics that are classification error and accuracy as follows:

$$\text{Classification error} := \frac{errors}{total}, \text{Accuracy=(1-error)} = \frac{correct}{total}$$

More precisely we can define the empirical error and the generalizzation error as follows:

- **True Error:** $error_{\mathcal{D}} = \mathbb{P}_{x \in \mathcal{D}}[c(x) \neq h(x)]$, where c is the true label and h is the predicted one.
- **Empirical Error:** $error_{Tr}(h) = \mathbb{P}_{(x,f(x)) \in Tr}[f(x) \neq h(x)] = \frac{|\{(x,f(x)) \in Tr : f(x) \neq h(x)\}|}{|Tr|}$

We can also now introduce the concept of overfitting: $h \in \mathcal{H}$ overfits Tr if $\exists h' \in \mathcal{H}$ such that $error_{Tr}(h) < error_{Tr}(h')$ but $error_{\mathcal{D}}(h) > error_{\mathcal{D}}(h')$.
In regression problems, the RMSE is the root of the mean of the squares of the deviations between the true value and the predicted value:

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(pred_i - true_i)^2}$$

Sometimes, we might use a confusion matrix to understand how the errors are distributed. A cell (r,c) reports the percentage of cases in which the system has predicted of class c a pattern of true class r. Ideally the matrix should be diagonal. High values off-diagonal indicate concentrarion of errors.

Open set

Let us consider a binary classification problem, where the two classes corresponds to positive (real class) and negative (rest of the world). We can consider only the positive class and a system capable of calculating the probability p of a pattern belonging to the class. Let t be the treshold value, then the pattern is classified as positive if $p > t$, as negative if not. We can noow define a **True Positive** as an outcome where the model correctly predict the positive class, and **True Negative** as an outcome where the model correctly predicts the negative class. A **False Positive** is an outcome where the model incorrectly predicts the positive class, while a **False Negative** is an outcome where the model incorrectly predicts the negative class. We can now define the following rate:

$$\text{False Positive Rate} = \frac{FP}{N_n}, \text{False Negative Rate} = \frac{FN}{N_p}$$

False positive and false negative rate are a function of the treshold t. High restrictive thresholds reduce false positives but increase the false negatives; vice versa, tollerant thresholds reduce false negatives but increase the false positives. We might also consider AUC (Area under the ROC curve), which is a measure of accuracy. The greater the AUC the greater the discriminating power of the approach. If AUC=1, we have a False Positive Rate = 0 and a True Positive Rate of 1 for all the values of the classification threshold.

Precision and Recall

We want to define two quantities:

$$\textbf{Precision} = \frac{t_p}{t_p + f_p}, \textbf{Recall} = \frac{t_p}{t_p + f_n}$$

The precision measure how many selected items are relevant, while recall measures how many relevan items are selected.

Multilabel Performance Indicators

- **One error:** evaluates the fraction that the label with the top-ranked predicted by the instance does not belong to its gorund truth relevant label set. The smaller the value of the one error, the better performance of the classifier.

$$\textbf{One-Error} = \frac{1}{n^t} \sum_{i=1}^{n^t} \mathbb{I}[\mathbf{y}^t_{ij^*} \neq -1]$$

- **Hamming loss:** evaluates the fraction of instance lable pairs which have been misclassifed. The smaller the value of the hamming loss, the better performance of the classifier.

$$\textbf{Hloss} = \frac{1}{n^t} \sum_{i=1}^{n^t} \frac{1}{q} \sum_{j=1}^{q} \mathbb{I}[y^t_{ij} \neq \hat{y}^t_{ij}]$$

Train, Validation, Test

The Training set is the set of patterns used to train the system, finding the optimal value for the parameters. The Validation set is the set of patterns on whih to set the hyperparameters H. The Test set is the set of patterns on whicch to evaluate the final system performance.

Convergence

The first aim to be pursued during the training is the convergence on the training set. Convergence occurs when the loss has a decreasing trend and Accuracy has an increasing trend. If the loss does not decrease, the system does not converge. It might be that the optimization model is not effective, the hyperparameters are out of range and so on. If the loss decreases but the accuracy does not increase, probably a wrong loss-function has been chosen. If the acccuracy does not approacc 100% on the training set, the degrees of freedom of the cllassifier are not sufficient to manage the complexity of the problem.

Bias-Variance Tradeoff

The bias error is an eerror from erroneous assumptions in the learning algorithm. HHHigh bias van cause an algorithm to miss the relevant relations betwee features and target outputs. It always lead to high error on training and test data. The variance is an error from sensitivity to small fluctuations in the training set. High variance can cause an algorithm to model the random noise in the training data, rather than the intended outputs. As a result, such models perform very well on training data but has high error rates on test data. There is the Bias-Variance decomposition from slide 99 to slide 100, i'm not gonna put this math.
Err(x) is the sum of $Bias^2$, variance and irreducible error. The latter is the error that can't be reduced by creating a good model. It is a measure of the amount of noise in our data. The sweetest spot we want to get is low variance and low bias. If our model is too simple and has very few parameters then it may have high bias and low variance. On the other hand if our model has large number of parameters then it's going to have high variance and low bias. To build a good model, we need to find a good balance between bias nad variance such that it minimizes the total error.

Neural Networks

Neurons

The first computational model of a neuron was the McCulloch-Pitts. The neruon was 'divided' into two parts: g which takes an input and performs an aggregation, and f which according to the aggregation created from g makes a decision. More formally the math is the following:

$$g(x_1, ..., x_n) = g(\mathbf{x}) = \sum_{i=1}^{n} x_i$$

The output is the following:

$$y = f(g(\mathbf{x})) = \begin{cases} 1 \ if \ g(\mathbf{x}) \geq \theta \\ 0 \text{ otherwise} \end{cases}$$

Activation Functions for shallow networks

The most used ones are:

- **Standard Logistic Function:** $f(net) = \frac{1}{q+e^{-net}}, f'(net) = f(net)(1 - f(net))$
- **Hyperbolic Tangent:** $f(net) = \frac{2a}{1+e^{-2bnet}} - a, f'(net) = \frac{4abe^{-2bnet}}{(a+e^{-2bnet})^2}$

Perceptron

We define the following parameters for the perceptron: $\Delta w_{ij} = \alpha(t_j - y_j)x_i$, where t_j is the target output, α is the learning rate, y_j is the output of the network and x_i is the input of the network. The update rule is the following: $w_i \leftarrow w_i + \Delta w_i$. The output of the system is given by the following trheshold function: $\begin{cases} 1 \text{ if } wx + b > 0 \\ 0 \text{ otherwise} \end{cases}$

Multilayer Perceptron

A Multilayer Perceptron (MLP) is a feedforward network with at least 3 levels and with non-linear acctivation functions. They are more expressive than liner models, since they can also learn function like XOR.

Forward Propagation

With forward propagation we mean the propagation of information forward: from the input level to the output level. Once trained, a neural network can simply process through forward propagation. The k-th output value can be calculated as:

$$z_k = f(\sum_{j=1,...,n_H} w_{jk}y_j + w_{0k})$$

Training

Once the topology is fixed, the training of the neural network consists in finding the value of the weights that determine the desired mapping between input and output. To hande a classification problem with s classes and d-dimensional patterns, it is usual to use a network with d input neurons, nh hidden neurons and s output neurons. A reasonable value for nh is d/10. Given a training pattern whose class is g, the desired output vecotr t takes the form: $t = [-1, ..., 1, ..., -1]$, where the 1 is for the chosen class. By choosing the sum of quares of the errors as the loss function, the error of pattern x is:

$$J(w, x) = \sum_{c=1,...,s} (t_c - z_c)^2$$

which quantifies how much the output produced for pattern x differs from the desired one. The aim of training is to minimize the objective function,i.e., the loss function. To do so, we use the gradient descent. Once we know how gradient descent works, we use backwards propagation to update the weights as follows:

$$w_{jk} = w_{jk} + \eta\delta_k y_j, \delta_k = (t_k - z_k)f'(net_k)$$

in case we are updating the weights form input to hidden layer, we substitute y_j with x_i.

Stocasti Gradient Descent

The most used approach for backpropagation. at each epoch, the n patterns of the training set are randomly ordered and then divided, considering them sequentially, into m groups of equal $size_{mb} = n/m$. The pseudocode is the following:

```
SGD
$n_h, m, w, maxEpoch, epoch \leftarrow 0$
do epoch← epoch+1
    random sort the Training Set
    for each mini-batch B of size n/m
        $grad_{ik} = 0.grad_{ij} = 0$
        for each x in B
            forward propagation di $x \rightarrow z_k, k = 1, ..., s$
            $grad_{jk} \rightarrow grad_{jk} + \delta_k y_j, k = 1, ..., s, j = 1, ..., n_h$
            $grad_{ij} \rightarrow grad_{ij} + \delta_k y_j, j = 1, ..., n_h, i = 1, ..., d$
        $w_{jk} = w_{jk} + \eta grad_{jk}, w_{ij} = w_{ij} + \eta grad_{ij}$
```

With epoch we mean the presentation of all the n patterns of the training et to the network. With iteration we mean the presentation of the patterns making up a mini-batch and the consequent update of the weights. $n/size_{mb}$ is the number of iterations per epoch. If it is not an integer, at the last iteration, fewer patterns are processed. The output function depends on the activation function applied at the output layer. We have no guarantee though that the sum on the output neuron is 1, so for multiple classes we use the softmax activation function, defined as follows:

$$z_k = f(net_k) = \frac{e^{net_k}}{\sum_{c=1,...,s} e^{net_c}}$$

Cross Entropy

For a multi-class classification problem, it is recommended to use Cross-Entropy as a loss function. The cross entropy betweenn two discrete distribution p and q is defined as:

$$H(p, q) = -\sum_{v} p(v) log(q(v))$$

When it's used as a loss function: p(fixed) is the target vector, while q is the output of the network. We can use the one-hot target to simpplify the use of the loss function ($t = [0, ..., 1, ..., 0]$):

$$J(w, x) = H(t, z) = -log(z_g)$$

if z_g is obtained by the softmax activation function then:

$$J(w, x) = -log(\frac{e^{net_g}}{\sum_{c=1,...,s} e^{net_c}}) = -net_g + log \sum_{c=1,...,s} e^{net_c}$$

Loss Regularization

We use loss regularization to reduce the risk of overfitting on the training set of a neural network with many parameters. To push the network to adopt weights of small value, a regularization term can be added to the loss: $J_{tot} = J_{classic} + J_{reg}$. We can have different types of regularization:

- **L2 Regularization:** $J_{reg} = \frac{1}{2}\lambda \sum_i w_i^2$
- **L1 Regularization:** $J_{reg} = \lambda \sum_i |w_i|$

In the case of SGD, the new weigth update would be of the form:

$$w_k = w_k - \eta(\frac{\partial J_{class}}{\partial w_k} + \lambda w_k)$$

Learning Rate

The calibration of the learning rate is an an optimal way is really importat. If it is too small we have slow convergence. On the other hand, if it is too big we can have a swing or divergence. The optimal value changes according to the network architecture of the loss function. A ccommonly employed technique calle learning rate annnealing, reccommens sarting wiht a relatively high learning rate and then graddually lowering the learning rate during training.

Cyclic Learning Rate

It is a technique to set, change and tweak the learning rate during training. It ims to train a NN with a LR that changes in a cyclical way for each batch. The bounds between whcih the LR will vary are base_lr and max_lr. We need to define the step_size,i.e. in how many epochs the learning rate will reach from one bound to another. The logic is that eriodic higher learning rates whithin each epoch helps to come out of any saddle points or local minima. The general schedule is:

$$\eta_t = \eta_{min} + (\eta_{max} - eta_{min})(max(0, 1-x)), x = |\frac{iterations}{stepsize} - 2(cycle) + 1|$$

where cycle is calculated as follows:

$$cycle = \lfloor (1 + \frac{iterations}{2(stepsize)}) \rfloor$$

Another cyclic LR, which allows us to get out of sharp minima is the foollowing:

$$\eta_t = \eta^i_{min} + \frac{1}{2}(\eta^i_{max} - \eta^i_{min})(1 + cos(\frac{T_{currrent}}{T_i}\pi))$$

Momentum

In SGD to avoid oscillations we can use momentum: the last update of each parameter is saved, and the new update is calucclated as a linear combination of the previous update and the current gradient. The typical value of the momentum(decay rate) is 0.9. We have the new update as:

$$\Delta w_{ij} = (\eta \frac{\partial E}{\partial w_{ij}}) + (\underbrace{\gamma}_{\text{momentum factor}} \Delta w^{t-1}_{ij})$$

Adam

Adam is an optimizer that computes adaptive learning rates for each parameter combining the ideas of momentum and adaptive gradient. Adam defines the first momentum and the second momentum as:

$$m_t = \rho_1 m_{t-1} + (1-\rho_1)g_t, u_t = \rho_2 u_{t-1} + (1-\rho_2)g^2_t$$

where g_t is the gradient at time t, the square of g_t is the component-wise square, ρ_1, ρ_2 are hyper-paraeters representing the exponential decay rate for the first and second momentum estimates (usually 0.9 and 0.999 respectively). We can define a bias-corrected version of the moving averages as $\hat{m_t} = \frac{m_t}{(1-\rho^t_1)}$ and $\hat{u_t} = \frac{u_t}{(1-\rho^t_2)}$. The final update for each θ_t parameter of the network is:

$$\theta_t = \theta_{t-1} - \lambda \frac{\hat{m_t}}{\sqrt{\hat{u_t}} + \epsilon}$$

where λ is the learning rate and ϵ is a really small positive number to prevent any division by zero.

diffGrad

It takes into account the difference of the gradient in order to set the learning rate. In order to define the update function, theu define the absolute diffference of two consecutive steps of the gradiient as $\Delta d_t = |g_{t-1} - g_t|$. The final update of each parameter for the sigmoid function is: $\theta_{t+1} = \theta_t - \lambda \xi_t \frac{\hat{m_t}}{\sqrt{\hat{u_t}+\epsilon}}$, where ξ is the sigmoid function.

Overfitting and DropOut

DropOut involves injecting noise while computing each internal layer during forward propagation, adn it has become a standard technique to train NN. We DropOut some neurons during training. In standard DropOut regularization, one debiases each layer by normalizing by the fraction of nodes that were retained. In other words, with dropout probability p, each intermediate activation h is replaced by a random variable h' as follows:

$$\begin{cases} 0 & \text{with probability p} \\ \frac{h}{1-p} & \text{otherwise} \end{cases}$$

Deep Convolutional Neural Networks

Networks made uoo if many levelr: at least two hidden layers. The nnumber of level, aka depth, is one of the complexity factors. Also number of neurons, connections and weights characcterize the complexity off a DNN. The two most known forward models for classification are Convolutional Neural Networks and fully connected Depp Neural Network.

Convolutional Neural Netwoks

They have been created due to the urge of reducing th ecomputational power behind large datasets. The training of complex models requires high computational power, but after training, the classification time is really low (even 100s of images per second on some powerful GPUs).
Other problems realted to DNN are the vanishing or exploding gradients:

- **Vanishing:** as the backpropagation algorithm advances backwards, the gradient often gets smaller and smaller and approaches zero which eventually leaves the weights of the initial or lower layers nearly unchanged. As a result, the gradient descent never converges to the optimum.
- **Exploding:** on other cases, the gradient keeps getting larger. It causes very large wigth updates and causes the gradient descent to diverge.

Gradients of unpredictable maggnityde threaten the stability of our optimization algorithms. We may be facing parameter updates that are iether excessively large or excessively small. Usually the vanishing gradient problem is caused by the choic eoff the activation function.

Parameter inizialization

One way to minimize the problems cited above is through careful inizialization of the parameters. To do so, we cna use the Xavier Inizialization:
Let us look at the scale distribution of an output o_i for some fully-connected layer *without non-linearities*. The output is given by: $o_i = \sum_{j=1}^{n_{int}} w_{ij}x_j$. The weights w_{ij} are all drwan independetly from the same distribution. Let's assume that this distribution has zero mean and variance σ^2. Through some calculation we'll obtain the following:

$$\mathbb{E}[o_i] = 0, \qquad Var[o_i] = n_{in}\sigma^2\gamma^2$$

where γ^2 is the variance.
One way to keep the variance fixed is to set $n_{in}\sigma^2 = 1$. We can see that if the gradients variance is not fixed as follows $n_{out}\sigma^2 = 1$ it can blow (n_{out} it the number of outputs of this layer). We are simply trying to satisfy:

$$\frac{1}{2}(n_{in} + n_{out})\sigma^2$$

Typically the Xavier initialization samles weights form a Gaussian distribution with zero mean and variance $\sigma^2 = \frac{2}{(n_{in}+n_{out})}$. We can also adapt Xavier's intuition to choose the variance when sampling weights from a uniform distribution. Note that the uniform distribution $U(-a,a)$ has variance $\frac{a^2}{3}$. By plugging this last result on the condition on σ^2 we obtain:

$$U(\sqrt{-\frac{6}{n_{in}+n_{out}}, \frac{6}{n_{in}+n_{out}}})$$

We obtained this result by imposing $\frac{a^2}{3} = \frac{2}{n_{in}+n_{out}}$. Even tho the assumption of nonexistence of nonlinearities can be violated in neural networks, the Xavier initialization method turns out to work well in practice.

Back to Convolutional Neural Neworks

The main difference with MLPs is the local processing: neurons are only locally connected to neurons of the previous level. It has significant reduction in the number of connections: weights are shared in groups. It has a strong reduction in the number of weights.
Convoluion is one of the most important operations in image processing. A digital filter is scrolled over the different input positions; an output value is generated for each position by running the calar product between the mask and the portion of the input covered. In case of multiple channels you do the same convolution per each channel and then sum the corresponding cells. We can use a 1x1 convolutional filter in order to ccorrelate channels between themselves.
we define as Volumes neurons organized in a 3D grid. The third dimension, the depth, identifies the different feature maps.
Filters operate on a portion of the input volume. Each slice of neurons denotes a feature map. Weights arre shared at the feature map level. Neurons in the same feature map process differentt portions of the input volume in the same way. Each feature map can be seen as the result of a specific input filter.
When a filter is scrolled down on the input volume, we can use a Stride. This operation reduces the size o the feature maps in the output volume and consequently the number of connections. Another option is to add a brder to the input volume. The Padding parameter denotes the thickness of the border. We can then express the size of the output as:

$$W_{out} = \frac{W_{in} - F + 2Padding}{Stride} + 1$$

where F is the size of the filter that is applied to the layer.

Activation Function

In MLP the most used activation function was the sigmoid. In Deep Networks, the use of the sigmoid is problematic for the backward propagation of the gradient. So, we needed new activation functions:

ReLU

f(net)=max(0,net), or more mathematically:

$$f(x)\begin{cases}0 \text{ for } x<0\\ x \text{ for } x\geq 0\end{cases} \qquad f'(x)\begin{cases}0 \text{ for } x<0\\ 1 \text{ for } x\geq 0\end{cases}$$

the derivative is zero for negative or zero values of net and 1 for positive values. For positive values no saturation. It leads to sparse activations which can confer greater robustness.

Leaky ReLU and ELU

The Leaky ReLU is defined as follows:

$$f(x_i) = \begin{cases} ax_i, & x_i < 0 \\ x_i, & x_i \geq 0 \end{cases} \quad f'(x_i) = \begin{cases} a, & x_i < 0 \\ 1, & x_i \geq 0 \end{cases}$$

No point where the gradient is null.
The ELU is defined as follows:

$$f(x_i) = \begin{cases} a(exp(x_i) - 1), & x_i < 0 \\ x_i, & x_i \geq 0 \end{cases} \quad f'(x_i) = \begin{cases} aexp(x_i), & x_i < 0 \\ 1, & x_i \geq 0 \end{cases}$$

PReLU

It is defined as follows:

$$f(x_i) = \begin{cases} a_c x_i, & x_i < 0 \\ x_i, & x_i \geq 0 \end{cases} \quad f'(x_i) = \begin{cases} a_c, & x_i < 0 \\ 1, & x_i \geq 0 \end{cases} \quad f'(x_i) = \begin{cases} x_i, & x_i < 0 \\ 0, & x_i \geq 0 \end{cases}$$

SReLU

The S-Shaped ReLU is composed of three piecewise linear functions that are expressed by four learnable parameters, thus:

$$f(x_i) = \begin{cases} t_i^r + a_i^r(x_i - t_i^r), & x_i \geq t_i^r \\ x_i, & t_i^r > x_i > t_i^l \\ t_i^l + a_i^l(x_i - t_i^l), & x_i \geq t_i^l \end{cases}$$

APLU

The Adaptive Piecewise Linear Unit is defined as follows:

$$f(x_i) = ReLU(x_i) + \sum_{c=1}^{n} a_c \max(0, -x_i + b_c)$$

Mexican ReLU

The Mexican ReLu(MeLu) is defined by the following parameters: $\phi^{a,\lambda}(x) = \max(\lambda - |x - a|, 0)$ be a mexican hat type function, where a and λ are real numbers. When $|x - a| > \lambda$, the phi function is null. It increases with a derivative of 1 between a-λ and a, while it decreases with a derivative of -1 between a and a+λ. We can now define the MeLU as:

$$y_i = MeLU(x_i) = PReLU^{c_o}(x_i) + \sum_{j=1}^{k-1} c_j \phi^{a_j,\lambda_j}(x_i)$$

where k is the number of learnable parameters for each channel, c_j are the learnable parameters, c_0 is the vector of parameters in PReLU, and a_j and λ_j are both fixed and chosen recursively.

GaLU

The Gaussian ReLU is based on MeLU and posseses the same desirable properties. To define GaLU, let $\phi_g^{a,\lambda} = \max(\lambda - |x-a|, 0) + \min(|x-a-2\lambda| - \lambda, 0)$ be a Gausssian type function, then Galu is defined as:

$$y_i = GaLU(x_i) = PReLU^{c_0}(x_i) + \sum_{j=1}^{k-1} \phi_g^{a_j,\lambda_j}(x_i)$$

Pooling

A pooling layer aggregates information in the input volume, resulting in smaller feature maps. The goal is to confer invaraince with respecct to simple tranformations of the input while maintainign significant infromation for pattern discrimination. It usually operates within each feature map, so that the number of feature maps in the input and output volume is the same.

softMax

It consists of s neurons fully-connected with repsect to the neurons of the previous level. The net_k activation level of the single neurons is caluclated the usual way, but as the activation function for the k-th neuron is:

$$z_k = f(net_k) = \frac{e^{net_k}}{\sum_{c=1,\dots,s} e^{net_c}}$$

where the values z_k produced van be interpeted as probabilities: they belong to $[0,1]$ and their sum is 1.

Cross Entropy- Loss Function

The cross entropy between two discrete distributions p and q is defined by:

$$H(p,q) = -\sum_v p(v) log(q(v)) \qquad J(w,x) = -log(z_g) = -log(\frac{e^{net_g}}{\sum_{c=1,\dots,s} e^{net_c}})$$

Gradients notes

Let's suppose we want to find the gradient of the following function:

$$C(y,w,X,b) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \max(0, w \cdot X_i + b))^2$$

Since it is a multivariable function, we need to find the partial derivatives of the function. To achieve so, we'll need different steps (some of them are omitted, so if you have doubts check the slides):
The neuron can be expressed as follows: $neuron(x) = \max(0, w \cdot x + b)$; thiss means that we want to calculate the derivative of the neuron w.r.t. the weights,i.e.

$$\frac{\partial neuron}{\partial w} = \frac{\partial neuron}{\partial z}\frac{\partial z}{\partial w}$$

As we can notice, z is composed by two parts: $w \cdot x$ and $+b$. The first one can be expressed as $\sum_i^n (w_i x_i) = sum(w \otimes x) := v$. we can then further define u:=sum(v). Now we can find the derivatives of v and u:

$$\frac{\partial v}{\partial w} = \frac{\partial}{\partial w}(w \otimes x) = diag(x) \qquad \frac{\partial u}{\partial v} = \frac{\partial}{\partial v} sum(v) = 1^{\to T}$$

Therefore, by putting everything together we obtain that $\frac{\partial u}{\partial w} = \frac{\partial u}{\partial v} = \frac{\partial v}{\partial w} = 1^{\to T} diag(x) = x^T$.
Now let's combine everything together and let us compute the derivatives of z=u+b w.r.t. w and b:

$$\frac{\partial z}{\partial w} = \frac{\partial u}{\partial w} + \frac{\partial b}{\partial w} = X^T + 0^{\to T} = x^T$$

$$\frac{\partial z}{\partial b} = \frac{\partial u}{\partial b} + \frac{\partial b}{\partial b} = \frac{\partial}{\partial b} w \cdot x + \frac{\partial b}{\partial b} = 0 + 1 = 1$$

What is the partial derivative of neuron(z) w.r.t. z? We know that Neuron(z)=max(0,z)=max(0, $sum($
$x) + b)$, so we can calculate the partial derivative as follows:

$$\frac{\partial}{\partial z} \max(0, z) = \begin{cases} 0 & z \leq 0 \\ \frac{dz}{dz} = 1 & z > 0 \end{cases}$$

so we will obtain that

$$\frac{\partial neuron}{\partial w} = \frac{\partial Neuron}{\partial z}\frac{\partial z}{\partial w} = \begin{cases} 0\frac{\partial z}{\partial w} = \vec{0}^T & z \leq 0 \\ 1\frac{\partial z}{\partial w} = \frac{\partial z}{\partial w} = x^T & z > 0 \end{cases} = \begin{cases} \vec{0}^T & w \cdot x + b \leq 0 \\ x^T & w \cdot x + b < 0 \end{cases}$$

Now we have everything we need to calculate the gradient of the loss function defined before, but first let's rewrite it as a composition of functions:

I $u(w, b, x) = \max(0, w \cdot x + b)$

II $v(y, u) = y - u$

III $C(v) = \frac{1}{N}\sum_{i=1}^N v^2$

The gradient w.r.t. the weight s is the one we calculated above, so we can now express the derivative of v as:

$$\frac{\partial v(y,u)}{\partial w} = \frac{\partial}{\partial w}(y - u) = \vec{0}^T - \frac{\partial u}{\partial w} = -\frac{\partial u}{\partial w} = \begin{cases} \vec{0}^T & w \cdot x + b \leq 0 \\ -x^T & w \cdot x + b > 0 \end{cases}$$

We can now compute, by using the chain rule, the derivative of the loss function w.r.t. the weights:

$$\frac{\partial C(v)}{\partial w} = \frac{\partial C}{\partial v}\frac{\partial v}{\partial w}$$

$$\frac{\partial C(v)}{\partial v} = \frac{\partial}{\partial v}\frac{1}{N}\sum_{i=1}^N v^2 = \frac{1}{N}\sum_{i=1} N\frac{\partial}{\partial v}v^2 = \frac{1}{N}\sum_{i=1}^N 2v$$

We know know both the derivative of C(v) and the derivative of v w.r.t. w. By multiplying the two derivatives togehter we obtain:

$$\frac{1}{N}\sum_{i=1}^N \begin{cases} 2v\vec{0}^T = \vec{0}^T & w \cdot x_i + b \leq 0 \\ -2vx^T & w \cdot x_i + b > 0 \end{cases}$$

By substituting v and u we would obtain:

$$= \frac{1}{N}\sum_{1=1}^{N}\begin{cases}\vec{0}^T & w\cdot x_i + b \le 0\\ -2(y_i - \max(0, w\cdot x_i + b))x_i^T & w\cdot x_i + b > 0\end{cases}$$

Since the max function is on the second line of our piecewise function, the max function will always output $w \cdot x + b$, so we can rewrrite the loss function as:

$$\frac{1}{N}\sum_{i=1}^{N}\begin{cases}\vec{0}^T & w\cdot x_i + b \le 0\\ -2(y_i - (w\cdot x_i + b))x_i^T & w\cdot x_i + b > 0\end{cases}$$

We can now see the $w \cdot x + b - y$ as an error term, so that we can write the loss function w.r.t. the weights as:

$$\frac{\partial C}{\partial w} = \frac{2}{N}\sum_{i=1}^{N} e_i x_i^T$$

This means that we can write the parameters update as follows: $w_{t+1} = w_t - \eta\frac{\partial C}{\partial w}$
What about the partial derivative w.r.t. the bias? The only thing that basically changes is the derivative of the function v(y,u):

$$\frac{\partial v(y,u)}{\partial b} = \frac{\partial}{\partial b}(y-u) = 0 - \frac{\partial u}{\partial b} = -\frac{\partial u}{\partial b} = \begin{cases}0 & w\cdot x + b \le 0\\ -1 & w\cdot x + b < 0\end{cases}$$

We can write now the derivative of the loss function as:

$$\frac{\partial C(v)}{\partial b} = \frac{\partial C}{\partial v}\frac{\partial v}{\partial b}$$

By substituting as in the previous case, we obtain the final system of eqautions as:

$$\frac{1}{N}\sum_{i=1}^{N}\begin{cases}\vec{0}^T & w\cdot x_i + b \le 0\\ 2(w\cdot x_i + b - y_i) & w\cdot x_i + b > 0\end{cases}$$

And just like before, we can epress an error term $e := w \cdot x + b - y$ and rewrite the loss function as:

$$\frac{\partial C}{\partial b} = \frac{2}{N}\sum_{i=1}^{N} e_i$$

So that the parameter update rule would be: $b_{t+1} = b_t - \eta\frac{\partial C}{\partial b}$

Some famous CNNs

CaffeNet

CaffeNet is a variant of AlexNet and it is composed by 7 hidden layers:

1. The input layer is a volume of 227x227x3.

2. The first and second layer are composed by a convolutional layer, a maxpooling layer and then a ReLU activation function.

3. The third and fourth layer have a convolutional filter and just the ReLU activation function.
4. The fifth layer has a convolutional filter, a max pooling and ReLU activation.
5. Layer 6 and 7 are fully connected and have convolutional filter and ReLU.
6. The output layer is a softmax

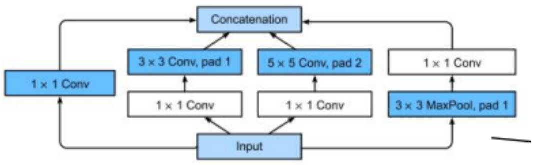
the stride is 4 for the first convolution, then it is always one. The convolutional filters have dimension 11x11 in the from input to first layer then always 3x3.It has about 60 Milions parameters.

EnzyNet

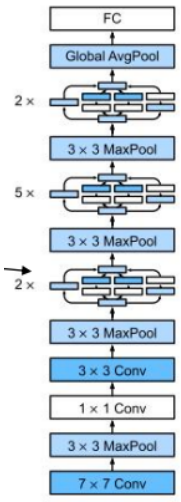
The EnzyNet takes input volumes of 32x32x32, then goes through a convolutional layer of 32 filters of size 9x9x9 with stride 2. then it goes through a second convolutional layer of 64 filters of size 5x5x5 witth stride 1. Is then followed by a max-pooling layer of size 2x2x2 with stride 2. Finallly, there are two fully-connected layers of 128 aand 6 hiddden units respectively. Everything then is concluded with a softmax layer tht outputs call probabilities.

GoogleNet

The basic convolutional block in the GoogleNet is called Inception Block, and it can be seen in the following image: It consists of four parallel paths. The first 3 paths use convolutional layers with



window sizes of 1x1, 3x3 and 5x5 to extract information from the number of channels, reducing the complexity of the model. The fourt path uses a 3x3 maximum pooling layer followed by a 1x1 convolutional layer to change the number of channels. Finally, all 4 paths are concatenated along the channel dimension and comprise the block's utput. The scheme of the whole network is the following:



ResNet

Batch normalization is applied to individual layers and it works as follows: in each training iteration, we first normalize the inputs by subtracting their mean and dividing by their standard deviation, where both are estimated based on the statistics of the current minibatch. Next, we apply a scale coefficient and a scale offset. More formally, denoting by $x \in \mathcal{B}$ an input to batch normalization that is from minibatch $\mathcal{B}$, batch normalization transforms x according to the following expression:

$$BN(x) = \gamma \odot \frac{x - \hat{\mu_{\mathcal{B}}}}{\hat{\sigma_{\mathcal{B}}}} + \beta$$

where $\hat{\mu_{\mathcal{B}}}$ and $\hat{\sigma_{\mathcal{B}}}$ are the mean and the standard deviation respectively. Nothe that γ, β are parameters to be learned jointly with other model parameters. Formally, we define the mean and the standard deviation as follows:

$$\hat{\mu_{\mathcal{B}}} = \frac{1}{|\mathcal{B}|} \sum_{x \in \mathcal{B}} \qquad \hat{\sigma_{\mathcal{B}}}^2 = \frac{1}{|\mathcal{B}|} \sum_{x \in \mathcal{B}} (x - \hat{\mu_{\mathcal{B}}})^2 + \epsilon$$

where epsilon is a small positive constant in order to avoid division by zero.
The ResNet is composed of the following layers: Batch normalization layers, fully connected layers, convolutional layers, and it emploies batch normalization during prediction.

Function Classes

Consider $\mathcal{F}$, the class of functions that a specific network architecture can reach. That is, for all $f \in \mathcal{F}$, there exists some set of parameters that can be obtained through training on a suitable dataset. Let us assume that f^* is the truth function that we really would like to find. If it is in $\mathcal{F}$, we are in good shape, but typically it won't happen. Instead, we'll try to find some $f^*_{\mathcal{F}}$ which is ur beest bet within $\mathcal{F}$. For instance, given a dataset with features X and labels y, we might try finding it by solving the following optimization problem:

$$f^*_{\mathcal{F}} := \arg\min_f L(X, y, f) \text{ s. t. } f \in \mathcal{F}$$

There are cases where the behavior is not expected, since we are moving through non-nested function classes. We'd like a Nested function classes case. More precisely, it is only reasonable to assume that if we design a different and more powerful architecture $\mathcal{F}$' we should arrive at a better outcome. In other words, we would expect that $f^*_{\mathcal{F}'}$ is better than $f^*_{\mathcal{F}}$. However, if $\mathcal{F} \not\subseteq \mathcal{F}$' there is no guarantee that this should even happen. In facct, $f^*_{\mathcal{F}}$.

Residual Blocks

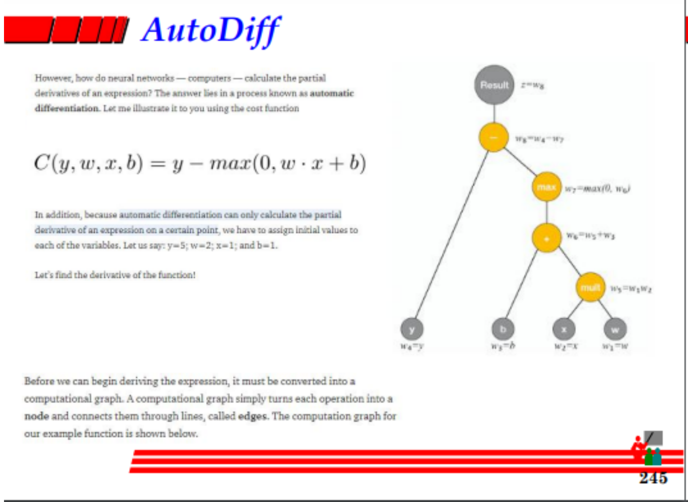
Let us focus now on a local part of a neural networks. Denote the input by x. Assume that after a layer we would obtain f(x). The residual block tries to learn the residual mapping f(x)-x, which is how the residual block derives its name. If the identity mapping f(x)-x is the desired underlying mapping, the residual mapping is easier to learn. We only need to push the weights and biases of the upper weigth layer within the block to zero.

DenseNet

It is a Resnet but is uses concatenation instead of residual block, so the input to the next block would be of the form:

$$x \to [x, f_1(x), f_2([x, f_1(x)]), f_3([x, f_1(x), f_2([x, f_1(x)])]), ...]$$

AutoDifferenziation



AutoDiff

However, how do neural networks — computers — calculate the partial derivatives of an expression? The answer lies in a process known as **automatic differentiation.** Let me illustrate it to you using the cost function

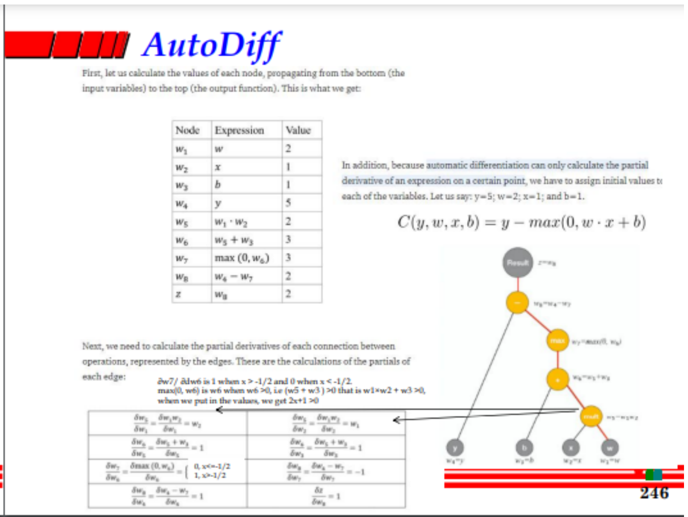
$$C(y, w, x, b) = y - max(0, w \cdot x + b)$$

In addition, because automatic differentiation can only calculate the partial derivative of an expression on a certain point, we have to assign initial values to each of the variables. Let us say: y=5; w=2; x=1; and b=1.

Let's find the derivative of the function!

Before we can begin deriving the expression, it must be converted into a computational graph. A computational graph simply turns each operation into a **node** and connects them through lines, called **edges**. The computation graph for our example function is shown below.

245



AutoDiff

First, let us calculate the values of each node, propagating from the bottom (the input variables) to the top (the output function). This is what we get:

Node	Expression	Value
w_1	w	2
w_2	x	1
w_3	b	1
w_4	y	5
w_5	$w_1 \cdot w_2$	2
w_6	$w_5 + w_3$	3
w_7	$\max(0, w_6)$	3
w_8	$w_4 - w_7$	2
z	w_8	2

In addition, because automatic differentiation can only calculate the partial derivative of an expression on a certain point, we have to assign initial values to each of the variables. Let us say: y=5; w=2; x=1; and b=1.

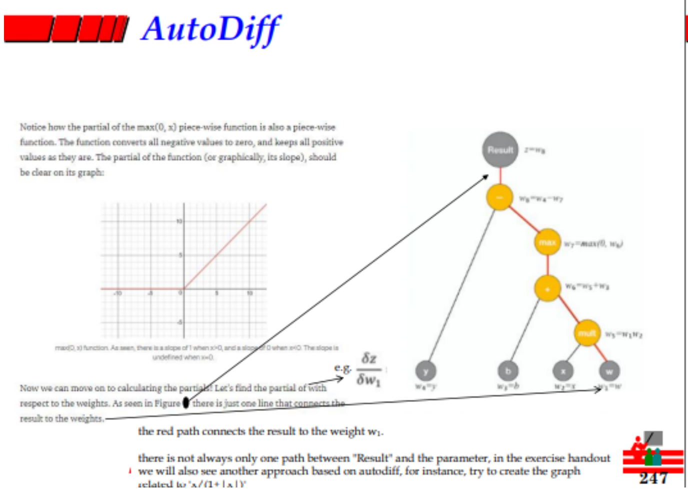
$$C(y, w, x, b) = y - max(0, w \cdot x + b)$$

Next, we need to calculate the partial derivatives of each connection between operations, represented by the edges. These are the calculations of the partials of each edge:

∂w7/ ∂dw6 is 1 when x > -1/2 and 0 when x < -1/2.
max(0, w6) is w6 when w6 >0, i.e (w5 + w3) >0 that is w1×w2 + w3 >0,
when we put in the values, we get 2x+1 >0

$\frac{\delta w_5}{\delta w_1} = \frac{\delta w_1 w_2}{\delta w_1} = w_2$	$\frac{\delta w_5}{\delta w_2} = \frac{\delta w_1 w_2}{\delta w_2} = w_1$
$\frac{\delta w_6}{\delta w_5} = \frac{\delta w_5 + w_3}{\delta w_5} = 1$	$\frac{\delta w_6}{\delta w_3} = \frac{\delta w_5 + w_3}{\delta w_3} = 1$
$\frac{\delta w_7}{\delta w_6} = \frac{\delta \max(0, w_6)}{\delta w_6} = \begin{cases} 0, x<=-1/2 \\ 1, x>-1/2 \end{cases}$	$\frac{\delta w_8}{\delta w_7} = \frac{\delta w_4 - w_7}{\delta w_7} = -1$
$\frac{\delta w_8}{\delta w_4} = \frac{\delta w_4 - w_7}{\delta w_4} = 1$	$\frac{\delta z}{\delta w_8} = 1$

246



AutoDiff

Notice how the partial of the max(0, x) piece-wise function is also a piece-wise function. The function converts all negative values to zero, and keeps all positive values as they are. The partial of the function (or graphically, its slope), should be clear on its graph:

max(0, x) function. As seen, there is a slope of 1 when x>0, and a slope of 0 when x<0. The slope is undefined when x=0.

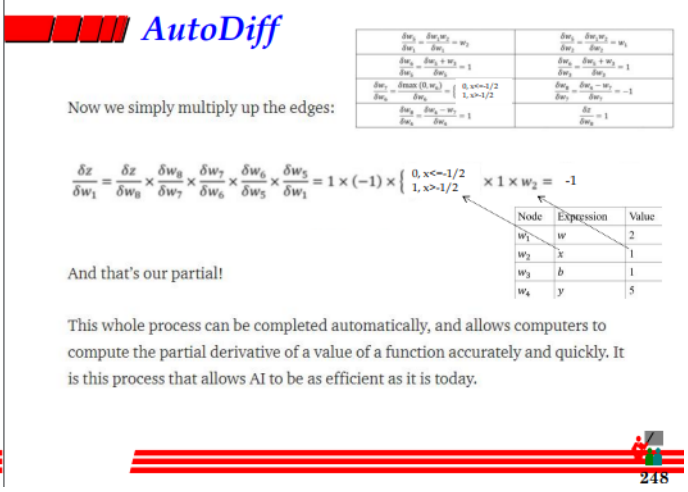
Now we can move on to calculating the partials! Let's find the partial of with respect to the weights. As seen in Figure ● there is just one line that connects the result to the weights.

e.g. $\frac{\delta z}{\delta w_1}$

the red path connects the result to the weight w₁.

there is not always only one path between "Result" and the parameter, in the exercise handout we will also see another approach based on autodiff, for instance, try to create the graph related to 'x/(1+|x|)'

247



AutoDiff

Now we simply multiply up the edges:

$\frac{\delta w_5}{\delta w_1} = \frac{\delta w_1 w_2}{\delta w_1} = w_2$	$\frac{\delta w_5}{\delta w_2} = \frac{\delta w_1 w_2}{\delta w_2} = w_1$
$\frac{\delta w_6}{\delta w_5} = \frac{\delta w_5 + w_3}{\delta w_5} = 1$	$\frac{\delta w_6}{\delta w_3} = \frac{\delta w_5 + w_3}{\delta w_3} = 1$
$\frac{\delta w_7}{\delta w_6} = \frac{\delta \max(0, w_6)}{\delta w_6} = \begin{cases} 0, x<=-1/2 \\ 1, x>-1/2 \end{cases}$	$\frac{\delta w_8}{\delta w_7} = \frac{\delta w_4 - w_7}{\delta w_7} = -1$
$\frac{\delta w_8}{\delta w_4} = \frac{\delta w_4 - w_7}{\delta w_4} = 1$	$\frac{\delta z}{\delta w_8} = 1$

$$\frac{\delta z}{\delta w_1} = \frac{\delta z}{\delta w_8} \times \frac{\delta w_8}{\delta w_7} \times \frac{\delta w_7}{\delta w_6} \times \frac{\delta w_6}{\delta w_5} \times \frac{\delta w_5}{\delta w_1} = 1 \times (-1) \times \begin{cases} 0, x<=-1/2 \\ 1, x>-1/2 \end{cases} \times 1 \times w_2 = \ -1$$

Node	Expression	Value
w_1	w	2
w_2	x	1
w_3	b	1
w_4	y	5

And that's our partial!

This whole process can be completed automatically, and allows computers to compute the partial derivative of a value of a function accurately and quickly. It is this process that allows AI to be as efficient as it is today.

248

Transfer Learning

Training complex CNNs on large datasets can take days or weeks of machine time, even if erfformed on GPUs. Fortunately, once the network has been trained, the time required for classifying a new pattern is usually fast. As alternative to training from scratch, we can pursue two transfer learning approaches: Fine-Tuning or Deepp Features.

Fine-Tuning

Fine-Tuning means starting with a pre-trained network trained on a similar problem and afterwards: Replace the output level with a new softmax output level; Same weights excepts for the penultimate and last level since they are intialized at random; New training iterations are performed to optimize the weights w.r.t. the peculiarities of the dataset.

Deep Features

Deep Features uses an existing netwokr without further fine-tuning. The features generated by the network during the forward step are extracted. These features are used to train an external classifier to classify the patterns of the new application domain.Usually it goes along PCA or DCT to reduce the dimensionality of the problem.

Reinforcement Learning

The goal is to learn optimal behavior from past experiences. An agent performs actions(a) that modify the environment, causing changes from one state(s) to another. When the agent obtains positive results, it receives a reward(r) which may be temporally delayed with respect to the action, or the sequence of actions which determined it. More formally we can define an episode(or game) as a finite sequence of states, actions and rewards:

$$s_0, a_0, r_1, s_1, a_1, r_2, s_2, ..., s_{n-1}, a_{n-1}, r_n, s_n$$

In each state $s_{t\text{-}1}$ the goal is to choose the optimal action $a_{t\text{-}1}$ that is the one that maximises the future reward $R_t := r_t + r_{t+1} + ... + r_n$. In many real applications there is a discount factor, so the reward becomes the following:

$$R_t = \sum_{i=t}^{n+t} \gamma^{i-t} r_t, \qquad 0 \leq \gamma \leq 1$$

We can also define it recursively as $R_t = r_t + \gamma R_{t+1}$.

Q-Learning

In Q-Learning, the Q(s,a) function indicates the optimality of the action a when it is applied at state s. The objective is to maximize the discounted future reward: $Q(s_t, a_t) = \max R_{t+1}$. Assuming that the function Q is known, when it is in state s, it can be shown that the optimal policy is the one that chooses action a^* such that: $a^* = \arg_a \max Q(s, a)$. The crucial point is therefore learning the Q function. Given a transition $< s_t, a_t, r_{t+1}, s_{t+1} >$ we can write:

$$Q = (s_t, a_t) = \max R_{t+1} = \max(r_{t+1} + \gamma R_{t+2})$$

$$\Rightarrow Q(s_t, a_t) = r_{t+1} + \gamma \max R_{t+2} = r_{t+1} + \gamma Q = (s_{t+1}, a_{t+1})$$

The action at t+1 will be chosen with the previous policy, obtaining:

$$Q(s_t, a_t) = r_{t+1} \gamma \max_a Q(s_{t+1}, a) \qquad \text{Bellman's equation}$$

In the algorithm(Slide 265), we have that α is the learning rate. if it is equal to 1, the update of $Q(s_t, a_t)$ is performed with Bellman's equation. If it is less than 1, the modification goes in the direction suggested by the Bellman's equation.

Sequence Based Neural Networks

The feedforward networks studied so far operate on input of a predetermined size. We woudl calle them one to one sequence. There are other type tho: one to many, Many to one and Many to many.

Recurrent Neural Networks(RNN)

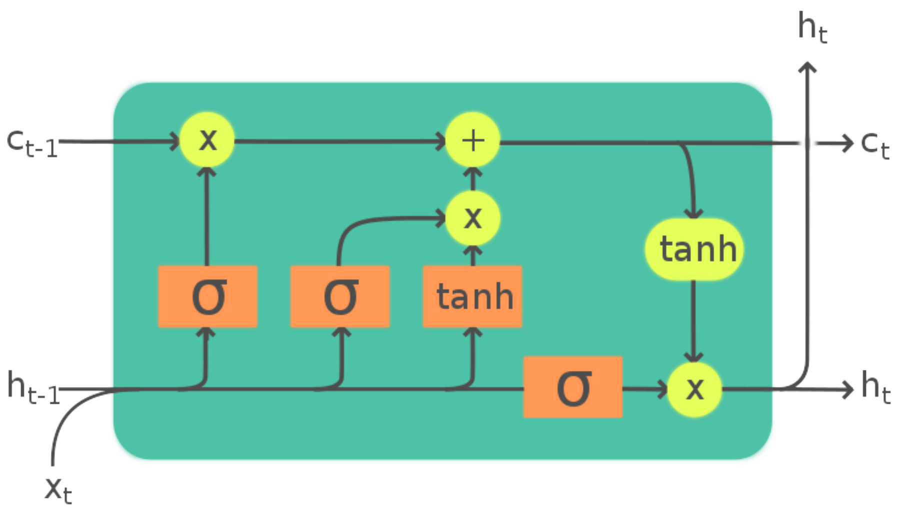
In recurring networks there are also backward connections or to the same level. The most common and widspreds models have connections towards the same level. At each step of the sequence in addition to the inut x_t, the level also receives the output from the previous stp y_{t-1}. This allows the network to make its decisions on past history or on all the elements of a sequence and on their reciprocal position.

Unfolding time

A cell is a part of a recurring network that preserves an nternal state h_t for each instant in time. It consists of a predetermined number of neurons. h_t depends on the input x_t and the previous state h_{t-1}. More formally $h_{(t)} = f(h_{(t-1)}, x_{(t)})$. In order to train an RNN it is necesary to perform the unfolding or unrolling in time and establishing a priori te number of time steps on which to perform the analysis. An unfolded RNN on 20 steps is equivalent to a feedforward DNN with 20 layers. Therefore, training RNNs can be very expensive and critical for convergence problem. This product of matrices is the source of exploding and vanishing gradients. Gradient clipping is a technique that tackles exploding gradients. The ides of gradient clipping is very simple: if the gradient gets too large, we rescale it to keep it small. More precisely, if $||g|| \geq c \Rightarrow g = c\frac{g}{||g||}$, where c is a hyperparameter, g is the gradient and $||g||$ is the norm of g.
In a base cell of RNN the state h_t depends on the input x_t and the previous state $h_{(t-1)}$: $h_{(t)} = \phi(x_{(t)} \cdot W_x + h_{(t-1)} \cdot W_h + b)$, where W_x and W_h are the weight vectors and b is the bias to be learned. ϕ is the activation function and the utput $y_{(t)} = h_{(t)}$. The base cells have diifficulty remembering/-exploiting inputs from distant steps: the memory of the first inputs tends to vanish. On the other hand, we know that in a sentence even the first words can have a very relevant importance. To solve this problem and facilitate convergence in complex applications, more advanced cells, with a long-term memory effect have been proposed: LSTM andd GRU are the best known topologies.

Long Short-Term Memory

LSTM is a RNN that makes a ddecision for what to remember at every time step. This network contains three gates: Input gate I, Output Gate O and a Forget Gate f, each of whicch consist of one layer with the sigmoid (σ) activation function. It also contains a specialized single layer



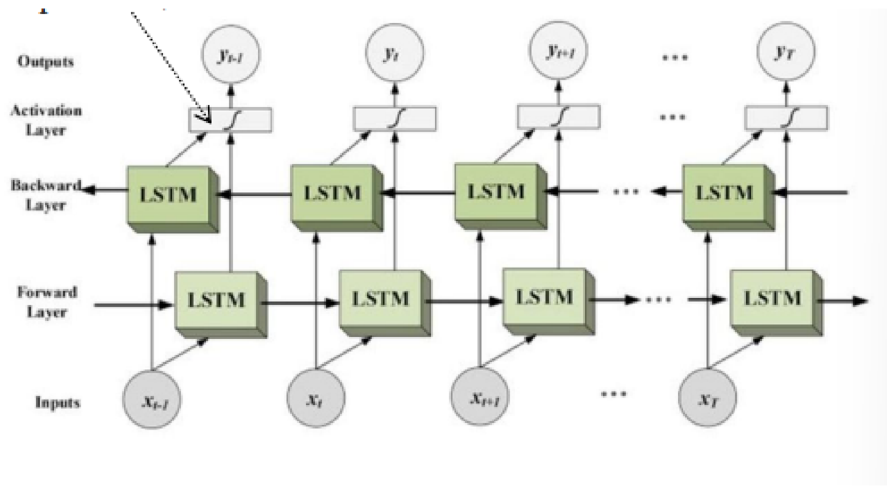
candidate $\bar{C}$, which has a Tanh activation function. In addition, there are four state vectors: memory state C, previous memory state C_{t-1}, an hidden state H with its previous state H_{t-1}. x_t is the input at time step t. The process for updating LSTM at time t is as follows: given X_t and H_{t-1} and letting U,W, b be the learnable weights of the network, the candidate layer $\bar{C}_t$ is

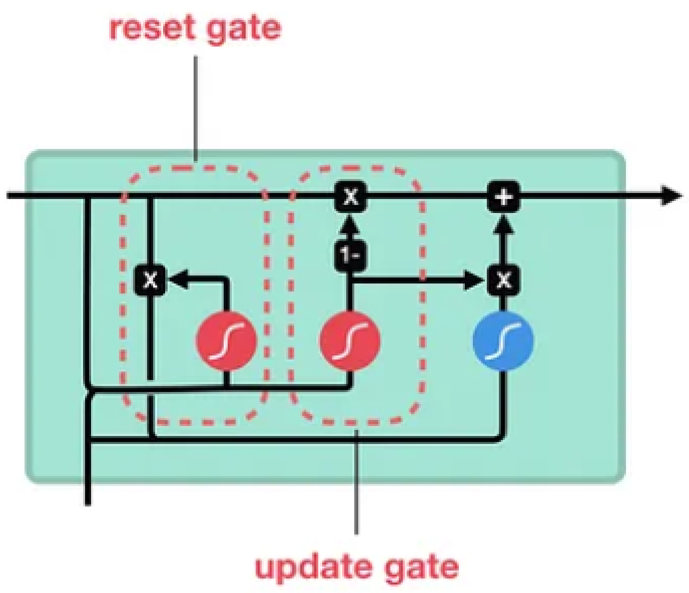
$$\bar{C}_t = \tanh(U_c X_t + W_c H_{t-1} + b_c)$$

The next memory cell is $C_t = f_t * C_{t-1} + I_t * \bar{C}_t$, where * os the element-wise multiplication. The gates are defined as follows:

$$f_t = \sigma(U_f X_t + W_f H_{t-1} + b_f) \qquad I_t = \sigma(U_i X_t + W_i H_{t-1} + b_i) \qquad O_t = \sigma(U_o X_t + W_o H_{t-1} + b_o)$$

The output is $H_t = O_t * \tanh(C_t)$. Regarding input, all sequences for this task are of the same legth, so sorting input by length is not required. An LSTM that has two stacked layers trained on the same set of samples is callde a Bidirectional LSTM (BiLSTM). The second LSTM connects to the end of the first sequencce and runs in revers. BiLSTM is best used to train data not related to time.





GRU

GRU can be considered as a simple LSTM variant. Differently from LSTM, GRU has a gate that lets the network decide which part of the old information is relevant to understand the new information. GRU has also fewer parameters and generally has a better performance on smaller data sets. The basic components of a GRU are reset gate and update gate: the first determines how much old information to forget, the second determines what information should forget and what information should pass to the output. Let x_t be the input sequence and initialize $h_0 = 0$. Then we define the update gate vector z_t and the reset gate vectors r_t as:

$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z) \qquad r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r)$$

where $W_z, U_z, b_z, W_r, U_r, b_r$ are matrices and vectors and σ is the sigmoid function. Then we define:

$$\hat{h}_t = \phi(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h)$$

to be the candidate activation vector, where ϕ is the tanh acivation function and $\odot$ is the component wise product. Notice that the term r_t determines the amount of past information that is relevant for the candidate activation vector. Then

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

is the output vector. The update gate vector z_t measures the amount of old and new information to keep. GRU (Gated Recurrent Unit) is a simplified version of LSTM, which tries to maintain its advantages, reducing parameters and complexity. The main simplifications are: only one state of memory h_t; a single gate to quantify how much to forget and how much to add.

Temporal convolutional network (TCN)

The main building block of a TCN is a dilated causal convolution layer, which operates over the time steps of each sequence. Causal means that the activations computed for a particular time

stepp cannot depend on activations from future time steps. To build up context from previous time steps, multiple convolutional layers are typically stacked on top of each other. To achieve large receptive field sizes, the dilation factor of subsequent convolution layers is increased exponentially. Assume that the dilation factor of the k-th convolutional layer is $2^{(k-1)}$ and the stridde is 1, then the receptive field size of such network can be computed as $R = (f-1)(2^K - 1) + 1$, where f is the filter size and K is the number of convolutional layers. Change the filter size and number of layers to easily adjust the receptive field size and the number of learnable parameters as necessary for the data and task at hand.
One of the disadvantages of TCNs compared to recurrent networks is that they have a larger memory footprint during inference. The entire raw sequence is required too compute the next time step. To reduce inference time and memory consumption, especially for stepp-ahead predicctions, train with the smallest sensible receptive field size R and perform prediction only with the last R time steps of the input sequence. The general TCN archtecture consists of multiple residual blocks, each containing two sets of dilated casual convolution layers with the same dilaction factor, followed by normalization, ReLU activation, and spatial dropout layers. The network addds the input of each block to the output of the block and applies a final activation function.

Recurrent DNN

We created a DNN architectiure based on GRU and TCN both adapted to a multilabel classification problem. The base schema of the model is a GRU with N hidden units, followed by a max pooling layer and a fully connected layer. The output layer is a sigmoid ffunction that provides multiclass classification. The TCN base approach has a similar architecture but with max pooling following the fully connected layer. The loss function is the Binary Cross-Entropy loss between the utputs and the actual labels. The multilabel Binary Crosss-Entropy loss calcculates the loss in the following way:

$$CELoss = -\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{l} y_i(j)\log(h_i(j)) + (1 - y_i(j))\log(1 - h_i(j))$$

where $y_i \in \{0,1\}^l$ and $h_i \in \{0,1\}^l$ are the atual and predicted label vectors of each sample $(i \in 1, ..., m)$, respecctively.

Generative Adversal Network

GAN are the networks used for the artificial generation of images or videos. It has two parts:

- The Generator learns to generate plausible data, The generated instances become negative training examples for the discriminator.
- Discriminator learns to distinguish the generator's fake data from real data. The discriminator penalizes the generator for producing implausible results.

When trianing begins, the generator produces obviously fake data, and the discriminator quickly leanrs to tell that it's fake.
Generative models try to generate very complex random variables. Supppose that we are interested in generating black and white quares oof dogs with a size of nxn pixels. We can reshape

each data as a N=nxn dimensional vecotr such that an image of dog can then be represented by a vector. However, it doesn't mean that all vectors represent a dog once shaped back to a square. So, we can say that the N dimensional vectors that effectively give something that look like a dog are distributed according to a very specific probability distribution over the entire N dimensional vecotr space. In the same spirit, there exists, over this N dimensional vector space, probability distributions for images of cats, birds and so on.
Then, the problem of generating a new image of dog is equivalent to the problem of generating a new vector following the dogg probability distribution over the N dimensional vector space. So, let's use transform method with a neural network as function.
Our first problem when trying to generate our new image of dog is that the dog probability distribution over the N dimensional vector space is a very complex one and we don't knwo how to directly generate complex randomm variables. However, as we know pretty well how to generate N uncorrelated unigorm random variables, we could make use of the transform method. To do so, we need to express our N dimensional random variable as the result of a very complex function applied to a simple N dimensional random variable.
As most of the time in these cases, very compex function naturally implies neural network modlling. Then, the idea is to model the transform function by a neural network that takes as input a simple N dimensional uniform random variable and that returns as output another N dimensional random variable htat should follow, after training, the right dog probability distribution. Once the architecture of the networrks has been designed, we still need to train it.
Training generative models requires a not directly compairison wwith true and generated distributions. We train the generative network by making these two distributions go through a downstram task chosen such that the optimization process of the generativve network with respect to the downstream task will enforce the generated distribution to be close to the true distribution.
We can use to train the networkthe fact that we can compare the probability distributions based on samples. We have a sample of true data and we can produce a sample of generated data. We can use MMD, which defines a distance between two probability distributions that can be computed based on samples of these distributions.
The idea of GMNs (Generative Matching Network) is to optimize the network by repeating the following steps:

- generate some uniform inputs
- make these inputs go trhough the network andd collect the generated outputs
- compare the true dog probability distributions and the generated one basaed on the available samples
- use backpropagation to make one step of gradient descent to lower the disstance between true and generated distributions.

Teh goal of the generator is to fool the discriminator, so the generativve neural network is trained to miximize the final classification error. The goal of the discrimator is to detect fake generated data, so the discriminative neurral network is trained to minimize the final classification error.
To learn the generator's distribution p_g over data x, we define a prior on input noise vvarriables $p_z(z)$, then represent a mapping space as $G(z, \theta_g)$, where G is a differentaible ffunction represented by a multilayer perceptron with parameters θ_g. We also define a second multilayer perceptron $D(x, \theta_d)$ that outputs a single scalar. D(x) represents the probability that x came from the data rather than p_g. We train D to maximize the probability of assigning the correct label to both

training examples and samples from G. We simultaneously train G to minimize $\log(1 - D(G(z)))$:

$$\min_G \max_D V(D, G) = \mathbb{E}_{x\sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))]$$

MvM

The capabilities of GAN in CV takes are limited by two main factors. These NN models distributions using statistical characteristics, such as mean and momens, rather than geometric characteristics. Secondly, traditional GANs rrepresent the losss of the discriminator network only in the form of a one-dimensional scalar value corresponding to the Euclidean distance between the real and fake data distributions.
In Maninfold Matching via Metric Learrning, two networks are trained against each other. The metric generator network learns to determine the best metric for the distribution generator network, and the distribution generator network learns to create negative examples for the metric generator network. Through competitive learning, MvM creates a distribution generator network that can create a fake data distriibution close to the real one, and a metric generator network that can probvide an effecctive metric to capture the internatl geometric structure of the dat distribution. Metric learning is an approach based ddirectly on distance metric that aims to estimate similarity or dissimilarity between imagees. Let G be a learnable pparameter, we define the distance between a and b as $dist(a, b) = ||Ga - Gb||_2$.

Siamese Network

A Siamese NN is a class of NN architectures that contain two or more identical subnetworks, which means they have the same configuration with the same parameters and weights. The algorithm compares the output of the upper neural network and the output of the lower neural network through a distance metric. To measure the distances of the two output we use the cosine similarity:

$$S_C(A, B) := \cos(\theta) = \frac{A \cdot B}{||A||||B||} = \frac{\sum_{i=1}^n A_i B_i}{\sqrt{\sum_{i=1}^n A_i^2}\sqrt{\sum_{i=1}^n B_i^2}}$$

With Triplet, we take tthree images as the inputs, labelled A, P and N. It is assumed that A and P have the same label and A and N different labels. In the training phase, for every Triplet in the training set, feature vectors F_A, F_P, F_N are computed then passed through a sigmoid to btain Y_A, Y_P, Y_N. At that point, the loss function is:

$$L = \max(|Y_A - Y_P|_2 - |Y_A - Y_N|_2, -\xi)$$

where ξ is a positive number and $|x|_2$ is the Euccliedean norm of the vector.

Segmentation

Image segmentation divides an image into several constituent regions. The methods for this problem usually make use of the correlation between pixels in the image. It does not need label information about image pixels during training, nd it cannot guarantee that the segmented segions will have the semantics that we hope to obtain during prediction.
Instance segmentation is also called simultaneous detection and segmentation. It studies how to recognize the pixel-level regions of each object instance in an image. Different from semantic segmentation, instance segmentation needs to distinguish not only semantics, but also different object

instances.
Region Growing is a reggion-based segmentation method, it involves the selection of initial seed points. This approach to segmentation examines neighboring pixels of initial seed points and determines whether the pixel neighbors should be added to the region. The process is iterated on, in the same manner as general data cluseting algorithms.

Segmentation Loss

Dice Loss is a metric widely used to estimate the performance of semantic segmentation models. The dice cofficient shows how similar two images are to each other. Its value is in $[0, 1]$. In order to appy dice loss to multiclass problems, the generalized dice loss was proposed. The formula o the generalized dice loss between the predictions Y and the training targets T is:

$$DL(Y,T) = 1 - \frac{2\sum_{k=1}^{K} w_k \sum_{m=1}^{M} Y_{km}T_{km}}{\sum_{k=1}^{K} w_k \sum_{m=1}^{M} Y_{km} + T_{km}} \qquad w_k = \frac{1}{(\sum_{m=1}^{M} T_{km})^2}$$

Where K is the number of classes and M is the number of pixels. The weighting factor w_k is introduced to help the network focuses on a small region. Indeed, it is inversely proportional to the frequency of the labels of a given class k. We can now define the Dice loss as:

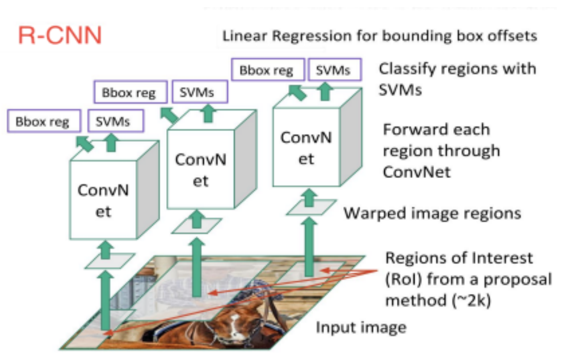
$$Dicd = \frac{2TP}{(TP + FP) + (TP + FN)}$$

A recurring problem in several image segmentation problem is the predominance of one class over another. In the interest of dealing with this issue, Tversky Loss was introduced, which derives from the Tversky Index. It has two weight factors α, β to manage trade-off between false positive and false negatives. When they are both 0.5 they degenerate into a dice similarity coefficient:

$$TI_c = \frac{TP}{TP + \alpha FN + \beta FP}, \qquad TL = \sum_{c=1}^{C} 1 - TI_c$$

where C is the number of classes.

Region based CNN (R-CNN)


R-CNN
Linear Regression for bounding box offsets
Bbox reg
SVMs
Classify regions with SVMs
Forward each region through ConvNet
ConvNet
Warped image regions
Regions of Interest (RoI) from a proposal method (~2k)
Input image

The R-CNN extraacts many region proposals from the input image, labeling their classes and bounding voxes. Then a CNN is used to perform forward propagation on eachh region proposal to eaxtract its features. Next, features of each region proposal are used for predicting the class and boudning box of this region proposal. The main performance bottleneck of a R-CNN lies in the independent CNN forward propagation for each reagion proposal, without sharing computation. Since these regions usually have overlaps, independent feature extractions lead to much repeated computation. One of the major improevements of the fast R-CNN from the R-CNN is that the CNN forward prpagation is only performed on the entire image. The steps are the following:

I. Perform selectivee search to extracct multiple high-quality region proposals on the input image different shapes and sies. Each region proposal will be labeled witha class and a ground-truth boundingg box.

II. Choose a pretrained CNN and truncate it before the output layer. Resize each region proposal to the input size required by the networks, and output the extracted features for the reggion proposal through forward propagation.

III. Take the extracted features and labeled class of each region proposal as an example. Train multiple SVMs to classify objects, where each SVMs individually determines wheter the example contains a specific class.

IV. Take the extracted features andd lavveled bounding box for each region proposal as an example. Train a linear regression model to predict the ground-truth boudning box.

To understand if the bounding box is correct we use the Intersection Over Union as a metric, defined as $IoU = \frac{Area\ of\ overlap}{Area\ of\ union}$. We keep only the bounding boxes that have an IoU less than 70%.

fast R-CNN

Intead of extracting CNN feature vectors independently ffor each region proposal, this model aggregates them into one CNN forward pass over the entire image and the region proposals share this feature matrix. Replace the last max pooling layer of the pre-trained CNN with a RoI poolingg layer, which outputs fixed-leength feature vectors for each region proposals. The RoI pooling layer is applied on the output of a chosen internal layer of the CNN. The R-CNN steps:

I. Compared witht the R-CNN, in the fast R-CNN the input of the CNN for fffeature extractions is the entire image, rather than idividual region proposals. Moreover, this CNN is trainable. Given an input image, let the shape of the CNN output be 1xcxh_1xx_1.

II. Suppose that the selective search generates n region proposals. Thesse region proposals on the CNN output. Then these regions of interest further extract features of the same shape in order to be easily concated. To achieve this, the fast R-CNN introduces the RoI pooling layer: the CNN output and region proposals are input into this layer, outputting concatenated features of shape $n \times c \times h_2 \times w_2$ that are further extracted for all the region proposals.

III. Using a fully-connected layer, transform the concatenated features into an output of shape nxd, where d depends on the model design.

IV. Predict the class and bounding box for each of the n region proposals. More concretely, in class and bounding box prediction, transform the fully-conneccted layer output into an output of shape nxq and an output of shape nx4, respecctively. The class prediction uses softmax reression.

To train the fast-R-CNN we use the following steps:

I. First, pre-train a CNN on image classification task

II. Propose reggions by sselective search

III. After the pretrained CNN: replace the last max pooling layer with RoI pooling layer; Replace the last fully connected layer and the last softmax layer with a fully connected layer and softmax over K+1 classes.

IV. Finally the model branches into two output classes: a softmax estimator of K+1 classes, outputing a discrete probability distribution per RoI; A bounding-box regression model which predicts offsets relative to the original RoI for each ot the K classes.

YOLO

It is significantly faster since YOLO does not divide the recognition into several phasses, but predicts bounding boxes, probabilities and classes of the object present in the input image in a single phase. It hase more localization errors, but at the same time it is less likely to recognize false positives in the background of the image,as well being considerably faster. The phases of YOLO execution: the model divvides the image into SxS cells, for each cell it predicts B boyìunding boox, each with a confidence score and C classes with relative probabilities. In Non Maxima Suppression, YOLO suppresses all bounding boxes that have lower probability scores.
To understand YOLO, it is necessary to establish what is actually being predicted. Ultimtely, we aim to predict a class of an object and the bounding box specifying object location. Each bouning bo can be described using ffour descriptors:

- center of a bounding box (bx, by)
- width (bw)
- height (bh)
- value c is the corresponding class

In addiction, we hav eto predicct the p_c value, which is the probability that there is an object in the bounding box.

Training phase

Fixed S andd b, the model divides the image into a grid of SxS cells. the grid cell containing the center of an object is defined as the cell responsible for it. Each cell ffo the grid predicts the B bounding boxes and assign each of them a score called Confidence Score. This score indicates how ikely it is that the bounding box contains an object and also how accurate the size and location of the bounding box with respect to the object is believed to be. Each cell of the grid, regardless of the number B of bouning boxes generated, predicts C classes with their conditional probability. At this point, the model multiplies the conditional probabilities of the classes with the confidence score of the bounding box, thus obtaining a specific confidence score of each class for eacch bounding box:

$$\mathbb{P}(Class_i|Object)\mathbb{P}(Object)IOU_{pred}^{truth} = \mathbb{P}(Class_i)IOU_{pred}^{truth}$$

The value thus obtained encodes both the probabilities that an object of a given class appears in the boundingg box under examination, and the precision with which the predicted bounding

box delimits the spatial boundaries of the object. The training is divided into two phases: a first pre-training phase, during which only the deepest levels are trained, and a second training phases that involves the entire network.
The final level of the network predicts both classes probabbilities and bounding box coordinatees for recognizing objects. The model normalized the width and height of the bounding boxes with respect to the size fo the input image so that their values fall in the real range between zero and one. The center of the bounding boxes is in turn normalized and expressed as a function of he cell to which it belongs so that it too belong to the interval between zero and 1. The YOLO loss function consists of classification, localization and confidence losses.

Neural Style Transfer

Neural Style Transfer is the action of appllying the style of an image to another image. The task needs two input images: one is the content image and the other is the style image. We will use NN to modify the content image to make it close to the style image in style. First, we initialize the synthetized image into the content image. This synthetized image is the only variable that needs to be updated during training. Then we choose a pretrained CNN to extract image features and freeze its model parameters during trianing. This deep CNN uses multiple layers to extract hierached features for images. We can choose the output of some of these layers as content fearures or style features.
Next, we calculate the loss function of style transfer through forward propagation and update the model parameters. the loss function commonly used consists of 3 parts: content, style and total variation loss. Finally, when the model training is over, we output the model parameters of the style transfer to generate the final synthetized image.
Let A^l_{ij} be the activation of the l-th layer, i-th feature map and j-th position obtained using the image l. Then the content loss is defined as :

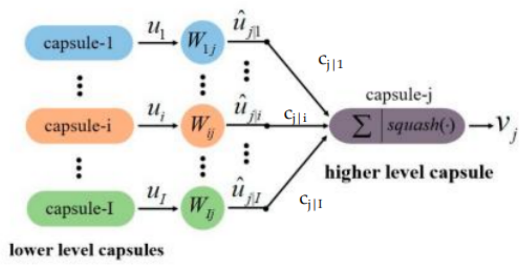
$$L_{content} = \frac{1}{2}\sum_{i,j}(A^l_{ij}(g) - A^l_{ij}(c))^2$$

In the stule transfer the only variable to be updated in the training step is the synthesized image.

$$L_{style} = \sum_{i}\sum_{i,j}(\beta G^{s,l}_{i,j} - \beta G^{p,l}_{i,j})^2$$

Capsule Network

Capsule Networks (CapsNets) are the networks that are able to fetch spatial infformation and more important features so as to overcome the loss of information that is seen in pooling operations. Capsules gives us a vector as an output that has a direction.
Froup of neurons tht perform a lo of internal computation and then encapsulate the results of these computations into a small vector of highly informative outputs. Inspired by mini-column in brain. Each capsule learns to recognize an impplicitly defined visual entity over a limited domain of viewing conditions and deformations. It ooutputs two things: the probaiblity that the enity is present within its limited domanin and a set of instantiation parameters, the generalized pose of the object.
Capsusles encode the probability of detection of a feature as the lenght of their output vector. And the state off the dected feature is encoded as the direction in which that vector points to. So when detected feature moves around the image or its state somehow changes, the probbability still stays the same,, but its orientation changes. This is what we refer to as activities equivariance: neuronal activities will change when an object moves over the manifold of possible appearences in the picture. at the same time, the probabilities off detection remains constant, which is the form of invariance that we should aim at, and not the type offered by CNNs with max pooling. Let



u_1, u_2, u_3 be the output vectors coming from capsules of the layer below. The vector is sent to all possible parents in the neural network. these vectors then are multiplied by corresponding weights matrices W that encode important spatial and other relationships betwee lower level features and higher level feature. W performs an affine transformation. We get the predicted position of the higher level feature $\hat{u}_{ij} = W_{ij}u_i$, i.e. where the face should be according to the detected position of the eyes. There are 4 main components that are present in the CapsNet that are listed below:

- Matrix Multiplication. it is applied to the image that is given as an input to the networrk to convert into vectors values to understand the spatial part.
- Scala Weigthin of the input. it computes which higher-level capsule should receive the current capsule output.
- Dynamic routing algorithm: it permits these different ccoomponents to trnasfer information amongst each other. Higher-level capsules get the input from the lower level. This is a iterative process.

- Squashing function: it is the last component that condenses the information. The squashing function takes all the information and converts it into a vector.

Vectors encode more information: relational and relative information. Capsules due to their richer vector information see that the sizes of the ffeatures are different and therefore output a lowe likelihood for the detection of the face. With this spatiaal information, we can detecct the inconsistences in the orienntation and size among the nose, eyes and ear features and therefore output a much lower activation for the face detection.
Affine Transformation: the input vectors represent either the initial input, or an input provided by an earlier layer in the networks. These vectors are first multiplied by the weight matrices. The weight matrix captures the spatial relationships. Say that one objcet is centerd around another, adn they are equally proportioned in size. The product of the input vector and the weight matrix will signify the high-level feature.

Dynamic routing

Weighting: a capsule nnetwork adjusts the wieghts such that a low-level capsule is strongly associated wiith high-level capsules that are in tis proximity. The proximity measure is determined by the afffine transformation step. The distance between the utputs obtained from the affine transformation step and the dense clusters of predictions of low-level capsules is computed. The high-level capsule that has the minimum distance beteen the cluster of already made predictions and the newly predicted one will have a higher weight, and the remaining capsule would be assigned lower weights, bassed on the distance metric. CapsNet it tries to send the information to the capsule abbov it that is best at dealing with it. The parameters W_{ij} models a part-whole relationship between the lower and higher level entitites.
We want the length of the output vvector of a capsule to represent the probability that the entity represented by the capsule is present in the current input. We therefore use a non-linear squashing function to ensure that short vectors get shrunk to almost zero length and lon vectors get shrunk to a length slightly below 1. The output vector of capsule j, with s_j is its total input is:

$$v_j = \frac{||s_j||^2}{1+||s_j||^2}\frac{s_j}{||s_j||}, \qquad s_j = \sum_i = c_{ij}\hat{u}_{j|i} \qquad \hat{u}_{j|i} = W_{ij}u_i$$

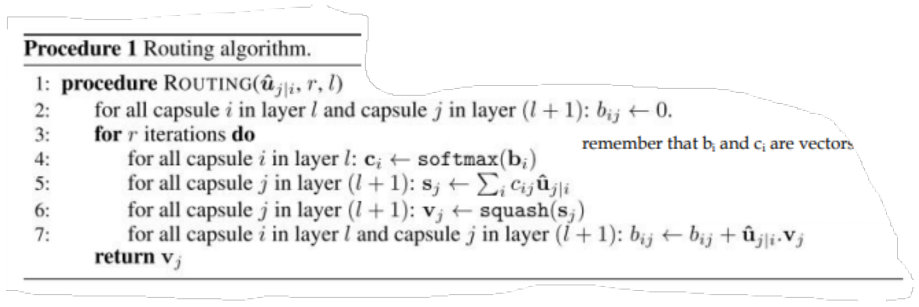
Where c_{ij} are the coupling coefficients that are determined by the iterative dynamic routing process. The coupling coefficients between capsule i and all the capsules in the layer above sum to 1 and are determined by a routing softmax whose initial logits b_{ij} are the log prir probabilities that capsule i should be cupled to capsule j:

$$c_{ij} = \frac{exp(b_{ij})}{\sum_k exp(b_{ik})}$$

The reouting algorithm is the following: The loss function is a separate margin loss for each capsule, for each category c of digit capsules:

$$L_c = T_c \max(0, m^+ - ||v_c||^2) + \lambda(1-T_c)\max(0, ||v_c|| - m^-)^2$$

where T_c is 1 if an object of class c is present. m+ is 0.9 and m- is 0.1, while $\lambda = 0.5$ down-weigthing for absent digit classes.



Procedure 1 Routing algorithm.

```
1: procedure ROUTING($\hat{\mathbf{u}}_{j|i}$, $r$, $l$)
2:     for all capsule $i$ in layer $l$ and capsule $j$ in layer $(l+1)$: $b_{ij} \leftarrow 0$.
3:     for $r$ iterations do
4:         for all capsule $i$ in layer $l$: $\mathbf{c}_i \leftarrow \mathtt{softmax}(\mathbf{b}_i)$
5:         for all capsule $j$ in layer $(l+1)$: $\mathbf{s}_j \leftarrow \sum_i c_{ij}\hat{\mathbf{u}}_{j|i}$
6:         for all capsule $j$ in layer $(l+1)$: $\mathbf{v}_j \leftarrow \mathtt{squash}(\mathbf{s}_j)$
7:         for all capsule $i$ in layer $l$ and capsule $j$ in layer $(l+1)$: $b_{ij} \leftarrow b_{ij} + \hat{\mathbf{u}}_{j|i}.\mathbf{v}_j$
       return $\mathbf{v}_j$
```

remember that b_i and c_i are vectors

Reconstruction as regularization method

Capsule networks use a reconstruction loss as a regularization method to encourage the digit capsule to encode the instantiation parameters of the input digit. In order to reconstruct the input from a lower dimensinal space, the Encoder and Decoder needs to learn a good matrix representation to relate the relationship between the latent space and the input.

Ensemble of networks

The idea is to train several models separately for the same task. At inference time average the results. The intuition is that different models make different errors on the test set. By averaging we obtain a more robust estimate withouth a better model. It works best if models are maximally uncorrelated. Winning entries of challenges are often ensembles, as empirically speaking is is very likely that useing ensemple methods give a 1-2% performance improvement in most tasks. The drawback is that it requires the evaluation of multiple models at inference time.

Q-statistic

Let $Z = \{z_1, ..., z_N\}$ be a labelled data set, $z_j \in \mathbb{R}^n$ coming fro the classification problem in question. For each classifier D_i we design an N-dimensional output vector $y_i = [y_{1,i}, ..., y_{N,i}]^T$ of corret classification, succh that $y_{i,j} = 1$ if D_i recognises correctly z_j, 0 otherwise. There are various statistics to assess the similarity of D_i, D_k. The Q statistic for two classifiers is:

$$Q_{i,k} = \frac{N^{11}N^{00} - N^{01}N^{10}}{N^{11}N^{00} + N^{01}N^{10}}$$

where N^{ab} is the number of elements z_j of Z for which $y_{j,i} = a$ and $y_{j,k} = b$. For statistically independent classifiers $Q_{i,k} = 0$ varies between -1 and 1. The correlatiion between two binary classifier outputs y_i and y_j is

$$\rho_{i,k} = \frac{N^{11}N^{00} - N^{01}N^{10}}{\sqrt{(N^{11} + N^{10})(N^{01} + N^{00})(N^{11} + N^{01})(N^{10} + N^{00})}}$$

For any two classifiers, Q andd ρ have the same sign, and it can be provede that $|\rho| \leq |Q|$.

Graph Convolutional Networks (GCN)

Classificaiton/Regression on Graphs

The dataset is composed of N pairs $\{(G_i, y_i), 1 \leq i \leq N\}$, n_i is a vertexm there is a discrete label associated to each node l(v). we have d vectorial attributes associated to each node: a(v) or $X \in \mathbb{R}^{n_j \times d}$. Given an unseen graph G, the task is to predict the corret target. The laplacian matrix L combines marices D and A as follows: L=D-A. it basiccaly means $L_{ij} = -A_{ij} = -w_{ij}$.

Graph Signal

Consider a given graph G with n nodes and shift operator S. A graph signal is a vector $x \in \mathbb{R}^n$ in which component x_i is associated with node i. To emphasize that the graph is intrinsic to the signal we may write the signa as a pair (S,x). The graph is an expecctation of proximity or similarity between components of the signal x. Define diffused signal y=SX$\Rightarrow$ $y_i = \sum_{j \in n(i)} w_{ij x_j}$. Codifies a local operation where components are mixed with components of neighbor nodes. Compose the diffusion operator to produce a diffusion sequence: $x^{k+1} = Sx^{(k)}, \qquad x^{(0)} = x$.
Graph convolution filters are the basic building bock of a GCN. Given S and filter coefficients h_k, a graph convolution is a polynomial on S:

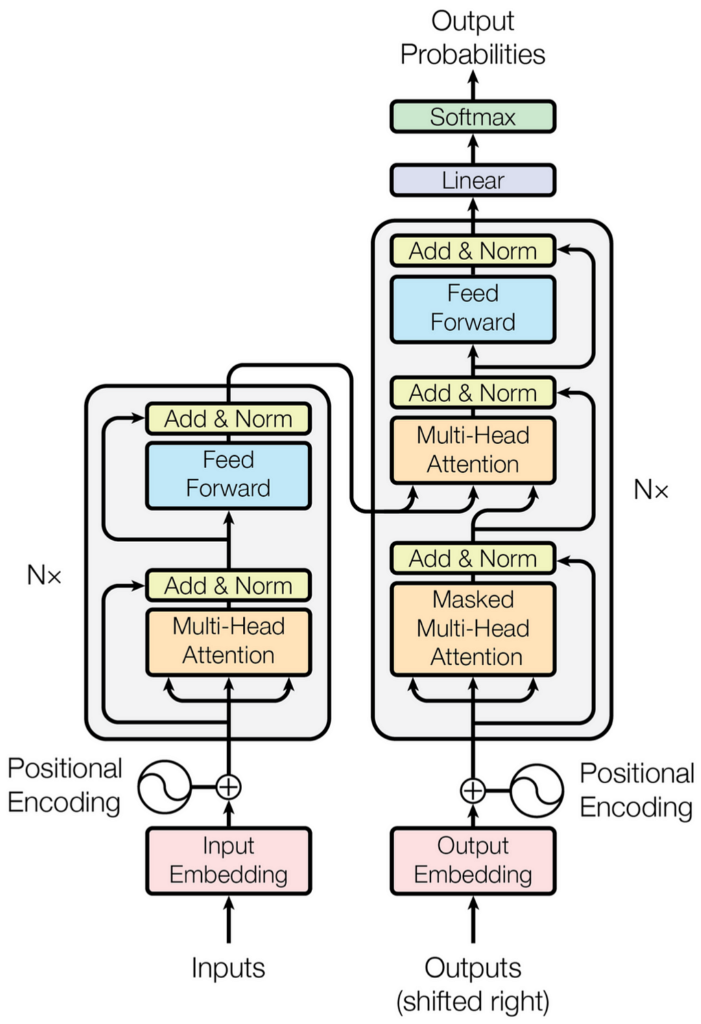
$$H(S) = \sum_{k=0}^{\infty} h_k S^k$$

Learning graph Filters: the prediction of a graph neural network is the following: $\hat{y} = f_h(x, S) = \sum_{k=0}^{K-1} h_k S^k x$. So we want to minimize the training loss L as:

$$h^* = \arg\min_h \sum_{(x,y,S) \in \mathcal{D}} \mathcal{L}(f_h(x, S), y)$$

Transformer

A language model models the probability distribution over a sequence of discrete tokens $x = (x_1, ..., x_T)$. Each of these tokens can take a value from a vocabulary $\mathcal{V}(x_t \in \mathcal{V})$ and it can be for example a word, a character or a byte, depending on the model.



Tokenization

Tokenization is the process of encoding a string of text into transformer-readable token ID integers. During inference/training, these token IDs are read by an embedding layer in our transformer model - which maps the token ID to a dense vector representation of that token.

Positional Encoding

officially, positional encoding is a set of small constants, which are added to the world embedding vector before the first self-attention layer. So, if the same word appears in a different position, the actual representation will be slightly different, depending on where it appears in the input sentence.

Key-Value-Query concept

When you search a query for a particular video, the search engine will map your query against a set of keys associated with possible sttored videos. Then the algorithm will present you with the best-matched values. This is the foundation of content/feature-based lookup. we splitt the data into key-valu pairs. We use the keys to define the attention weights to look at the data and the values as the information that we will acctually get. For the so called mapping, we need to quantify similarity, that we will be seeing next.
In geometry, the inner vector product is interpreted as a vector projection. One way to define vector similarity is by computing the normalized inner product.

Self attention

It is an attention mechanism relating different positions of a single sequencce in order to compute a representation of the sequence. It enables us to find correlations between different words of the input indicating the syntactic and contextual structure of the sentence. Having the Query, Value and Key matrices, we can now apply the self-attention layer as

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

Multi-head attention

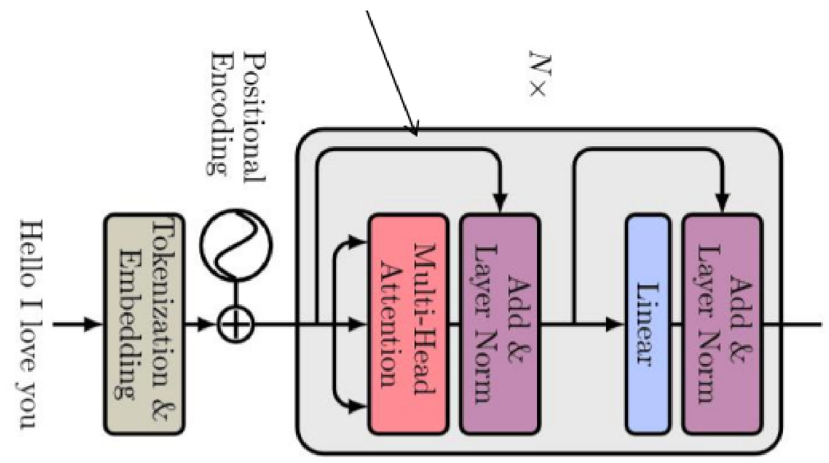
we run through the attention mechanism several times. Each time, we map the independent set of Key, Query, Value matrices into different lower dimensional spaces and compute the attention there. The mapping is achieved by multiplying each matrix with a separate weight matrix, denoted as $W_i^K, W_i^Q \in \mathbb{R}^{d_{model} \times d_k}$ and $W_i^V \in \mathbb{R}^{d_{model} \times d_k}$.

Normalization Layer

In the normalization Layer, the mean and suqare variaance are computed across channels and spatial dimensions. In language, each word is a vector. Since we are dealing with vectors, we only have one spatial dimention. In this case, the normalization is done througgh columns (top-bottom), instead then through rows.
It then follows a linear layer.

Encoders

To process a sentence we need three steps: 1, word embeddings of the input sentece are ccomputed simultaneously; 2, positional encodings are then applied to each embedding resulting in word vectors that also include positional information; 3, the word vectors are passed to the first encoder block. Each block consists of the following layers in the same order: a multi-head self-attention layer

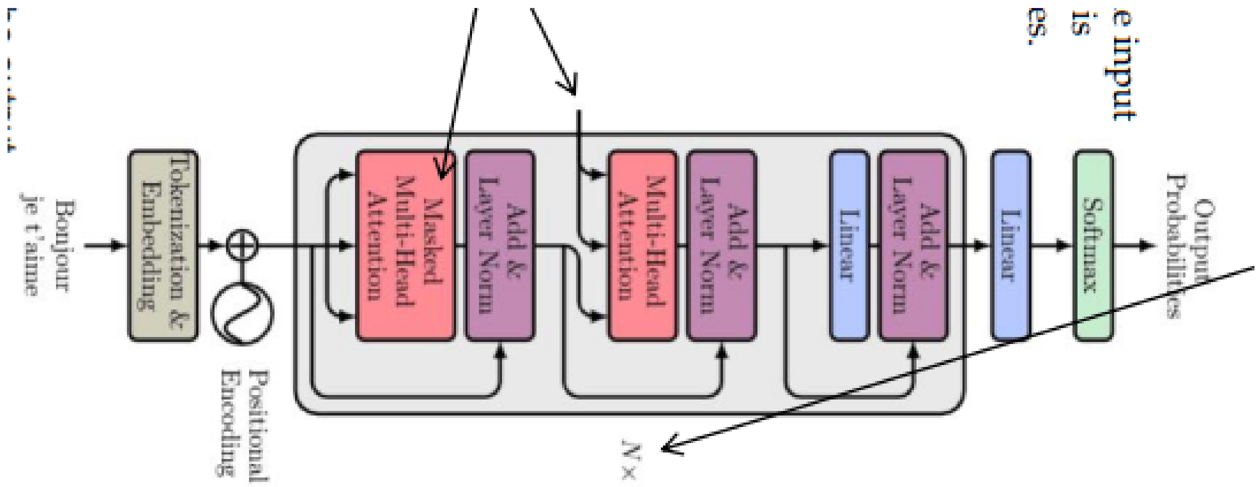


to find correlations between each word; a normalization layer; a residual connection around the prevvious two sublayers; a linear layer; a second normalization layer; a second residual connection.

Decoder

The decoder consists of all the aferomentiod components plus two novel ones.

Encoders



As before, the output sequence is fed in its entirety and word embeddings are computed; positional encodings are again applied; the vectors are passed to the first decoder block. Each encoder block icludes: a Masked multi-head self-attention layer; a normalization layer followed by a residual connection; a new multi-head attention layer; a second normalization layer and a residual connection; a linear layer and a third residual connection.

Output

The output probabilities predict the next token in the output sentence. We assign a probability to each word and we simply keep the one with the highest score. while most concepts of the decoder are already familiar, there are two more that we need to discuss. Let's start with the masked multi-head self-attention layer.

Masked multi-head attention

In the decoding stage, we predict one word(token) after another. In such NLP problems, sequential token predictions is unavoidable. As a result,the selff attentin layer need to be modified in order to consider only the output sentence that has been generated so far. Mathematically we have:

$$MaskedAttention(Q,K,V) = softmaxe(\frac{QK^T + M}{\sqrt{d_k}})V$$

Where the matrix M consists off zeros and -inf. Zeros will become ones with the exponential, while -inf becomes zeros. This effectively has the same effect as removing the corresponding cconnection. The remaining principles are exactly the same as the encder's attention. And once again, we can implement them in parallel to speed up the computations. Obviously, the mask wiill change for every new token we compute.

Encoder-Decoder Attention

This is acctually where the decoder processe the enccoded representation. The attention matrix generated by the encoder is passed to another attention layer alongside the result of the previous Masked multi-head attention block. The intuition behind the encoder-decoder attention layer is to combine the input and output sentence. The encoder's output encapsulates the final embedding of the input sentence. It is like our database. So we will use the necoder ooutput to produce the Key and Value matrices, On the other hand, the output of the masked Multi-head attention block contains the s far generated new sentence and is represented as the Query matrix in the attention layer. Again, it is the search in the database. Notice that the output of the last block of the encder will be usead in each decoder block.