

Deep learning

Intro

Main uses of SSL as of 2021

Two Uses for Self-Supervised Learning	Y. LeCun
<ul style="list-style-type: none">▶ 1. Learning hierarchical representations of the world<ul style="list-style-type: none">▶ SSL pre-training precedes a supervised or RL phase▶ 2. Learning predictive (forward) models of the world<ul style="list-style-type: none">▶ Learning models for Model-Predictive Control, policy learning for control, or model-based RL.▶ Question: how to represent uncertainty & multi-modality in the prediction?	

Why we need many hidden layers? Why we need deep learning?

1. Two layers NN are universal approximators but extremely inefficient to train
2. Deep ANNs transform the space nonlinearly so that the dataset becomes linearly separable
3. The larger the number of layers the larger the generalization of the embeddings
4. In overparameterized problems it is much easier to find a minimum
5. The world is compositional

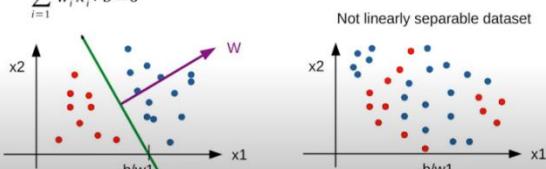
1. Two layers NN are universal approximators but

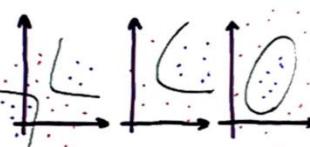
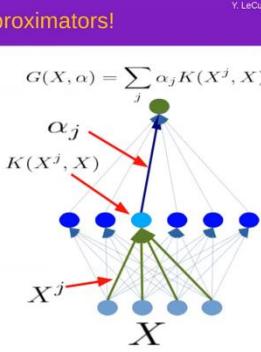
A theorem says that 2 layers NN are universal approximators as long as you have a large number of neurons in the hidden layer. They can approximate any function. The reason though that we need NN with more than one hidden layer, is that there are many functions that are really inefficient to approximate with only one hidden layer. You would need a ridiculous number of units in the hidden layer which will make training extremely inefficient. This is the reason we need deep learning. It took the ML community decades to understand this. It became obvious in the beginning of 2010s.

2. Deep ANNs transform the space nonlinearly so that the dataset becomes linearly separable

In a two way classification problem, if the number of training examples is larger than the number of features then the chances the decision boundary to be linear is almost zero. This is an old theorem (1966). So we need to find way to deal with non linear decision boundaries. So you have to somehow transform the feature space *non linearly*, so that in the new feature space, the decision boundary is linear. What you actually do is that you expand the space non linearly, you increase the number of dimensions of the feature space with non linear dimensions let's say, so that in the new space, the dimensions (number of features) is larger than the number of datapoints. This makes it easy to find a linear decision boundary.

This is what manual feature extraction tried to do back in the day. To transform the features in such a way that they can be separated linearly. The important thing is that this preprocessing must be non linear. Deep networks do this non linear feature creation automatically. For intuition of why this is the case, check [what ANNs actually do](#) chapter.

Linear Classifiers and their limitations	
<ul style="list-style-type: none">▶ Linear classifier $\bar{y} = \text{sign}(\sum_{i=1}^N w_i x_i + b)$▶ Partitions the space into two half spaces separated by the hyperplane: $\sum_{i=1}^N w_i x_i + b = 0$	Linear classifiers (one unit ANN, SVM, etc.) were typically used in supervised learning. A linear classifier in this case is a system with just one unit/neuron (and the input units of course). Have in mind this intuition about the decision boundary of such a linear classifier. The weight vector w is orthogonal to the decision boundary because the weighted sum is a dot product between w and x . two vectors have dot product of 0 when they are orthogonal. When the weighted sum is 0 (b is just an offset) we are at the decision boundary because the system outputs 0. (b changes the position, w changes the orientation)

<p>Ideas for "generic" feature extraction</p> <ul style="list-style-type: none"> ▶ Basic principle: ▶ expanding the dimension of the representation so that things are more likely to become linearly separable. ▶ - space tiling ▶ - random projections ▶ - polynomial classifier (feature cross-products) ▶ - radial basis functions ▶ - kernel machines 	<p>For non linearly separable cases, you have to transform space non linearly (expanding the space non linearly increasing the number of dimensions so that the number of dimensions is larger than the number of examples and the dataset has a chance of being linearly separable), so that in the resulting space the data set is linearly separable. This is what polynomial regression does. Or kernel machines in SVM.</p>
<p>Example: monomial features</p> <ul style="list-style-type: none"> ▶ Feature extractor computes cross products of input variables ▶ A linear classifier on top computes a polynomial of input variables $\Phi(x_1, x_2) = [1, x_1, x_2, x_1x_2, x_1^2, x_2^2]$ <ul style="list-style-type: none"> ▶ generalizable to degree d ▶ Unfortunately impractical for large d ▶ Number of features is d choose N, which grows like N^d ▶ But $d=2$ is used a lot in "attention" circuits. 	<p>polynomial regression for space transformation</p> <p>Have in mind that what polynomial regression actually does, is to transform the feature space in a non linear way, so that after the transformation, in the new space, the datapoints are linearly separable and we can use a typical linear classifier. We add more features (x_1x_2, x_1^2, x_2^2) and we do linear classification in that space which is a quadratic classification in the original space.</p>
<p>Shallow networks are universal approximators!</p> <ul style="list-style-type: none"> ■ SVMs and Kernel methods <ul style="list-style-type: none"> ▶ Layer1: kernels; layer2: linear ▶ The first layer is "trained" with the simplest unsupervised method ever devised: using the samples as templates for the kernel functions. ■ 2-layer neural nets <ul style="list-style-type: none"> ▶ Layer1: dot products + non-linear function; Layer2: linear ▶ But few useful functions can be efficiently represented with only two layers of reasonable size. 	<p>SVM with kernels for space transformation</p> <p>SVM with kernels as ANN</p> <p>You can think of SVM with kernels as an ANN with one hidden layer where the number of units are the same with the number of training examples and we train the network in such a way that a unit produces 1 or close to it if the input vector is the same as a training example. Each unit shows the similarity with a different training example. (they are the landmarks). But it has only one hidden layer with all the problems this brings. (no deep learning).</p>

3. The larger the number of layers the larger the generalization of the embeddings

The larger the number of layers the smaller the layers need to be in order to approximate a specific function. Smaller layers, mean smaller embeddings which can make them more generic. They can be enforced to capture only the important aspects of the input, the most important patterns. So they can generalize better. Have in mind that the ability of a system to generalize depends on its architecture that must be suitable for the data it receives (convnets for vision, transformers for speech).

4. In overparameterized problems it is much easier to find a minimum

In overparameterized problems it is much easier to find a minimum using optimization methods. This is the reason why many ANNs use more parameters than you might think is necessary. (one intuition I guess is that it is easier to overcome local minima in certain dimensions (certain parameters) by moving towards the other dimension. So the optimization algorithm can have many options for escaping local minima.)

5. The world is compositional

So this multilayer architecture with end to end learning allows the ANNs to learn hierarchies of simple to complex patterns. Also this compositionality makes the world comprehensible in the sense that we can process high level concepts that represent complex patterns and we can reason with these high concepts.

History

have in mind module based automatic differentiation

you can have modules of ANNs and if you know how to propagate gradients through these modules, you can compute gradients for any kind of graph of interconnected modules (assembly of modules) creating complex systems.

Inspiration for Deep Learning: The Brain!

- ▶ 1943: McCulloch & Pitts, networks of binary neurons can do logic
- ▶ 1947: Donald Hebb, Hebbian synaptic plasticity
- ▶ 1948: Norbert Wiener, cybernetics, optimal filter, feedback, autopoiesis, auto-organization.
- ▶ 1957: Frank Rosenblatt, Perceptron
- ▶ 1961: Bernie Widrow, Adaline
- ▶ 1962: Hubel & Wiesel, visual cortex architecture
- ▶ 1969: Minsky & Papert, limits of the Perceptron



More History

- ▶ 1970s: statistical pattern recognition (Duda & Hart 1973)
- ▶ 1979: Kunihiko Fukushima, Neocognitron
- ▶ 1982: Hopfield Networks
- ▶ 1983: Hinton & Sejnowski, Boltzmann Machines
- ▶ 1985/1986: Practical Backpropagation for neural net training
- ▶ 1989: Convolutional Networks
- ▶ 1991: Bottou & Gallinari, module-based automatic differentiation
- ▶ 1995: Hochreiter & Schmidhuber, LSTM recurrent net.
- ▶ 1996: structured prediction with neural nets, graph transformer nets
- ▶
- ▶ 2003: Yoshua Bengio, neural language model
- ▶ 2006: Layer-wise unsupervised pre-training of deep networks
- ▶ 2010: Collobert & Weston, self-supervised neural nets in NLP

Y. LeC

More History

- ▶ 2012: AlexNet / convnet on GPU / object classification
- ▶ 2015: I. Sutskever, neural machine translation with multilayer LSTM
- ▶ 2015: Weston, Chopra, Bordes: Memory Networks
- ▶ 2016: Bahdanau, Cho, Bengio: GRU, attention mechanism
- ▶ 2016: Kaiming He, ResNet

Future challenges

There are no systems as of 2021 that can do these things well with a few exceptions maybe.

1. Self supervision (no labels)
2. Reasoning (memory network was and is an attempt) reasoning with learning. A lot of people working on this
3. Autonomous agents that can plan based on objectives and model of the world. We don't know yet how to train a system to represent action plans in a hierarchical manner.

- ▶ How do we get DL systems to:
- ▶ Use a working memory?
- ▶ Perform long chains of reasoning?
- ▶ Remember massive amounts of factual knowledge?
- ▶ Plan complex sequences of actions?
- ▶ Learn hierarchical representations of action plans?

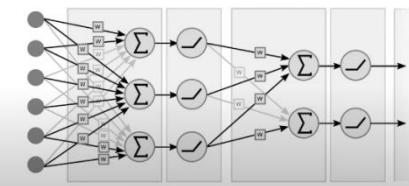
Notation

Block diagram notations for computation graphs

- ▶ Variables (tensor, scalar, continuous, discrete...)
 - ▶ Observed: input, desired output...
 - ▶ Computed variable: outputs of deterministic functions
- ▶ Deterministic function
 - ▶ Multiple inputs and outputs (tensors, scalars,....)
 - ▶ Implicit parameter variable (here: w)
- ▶ Scalar-valued function (implicit output)
 - ▶ Single scalar output (implicit)
 - ▶ used mostly for cost functions

Traditional Neural Net

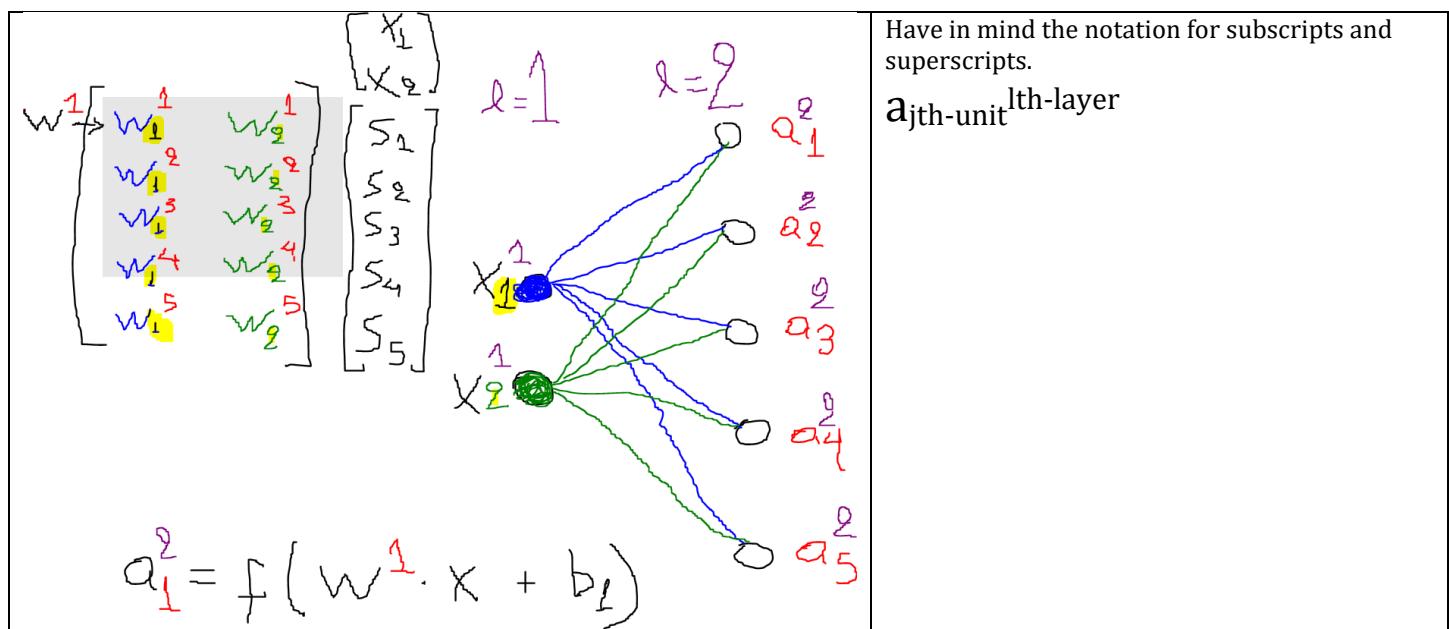
- ▶ Stacked linear and non-linear functional blocks
- ▶ Weighted sums, matrix-vector product
- ▶ Point-wise non-linearities (e.g. ReLu, tanh,)



Grey background in the input means it is an observation
Red circle means learned parameter

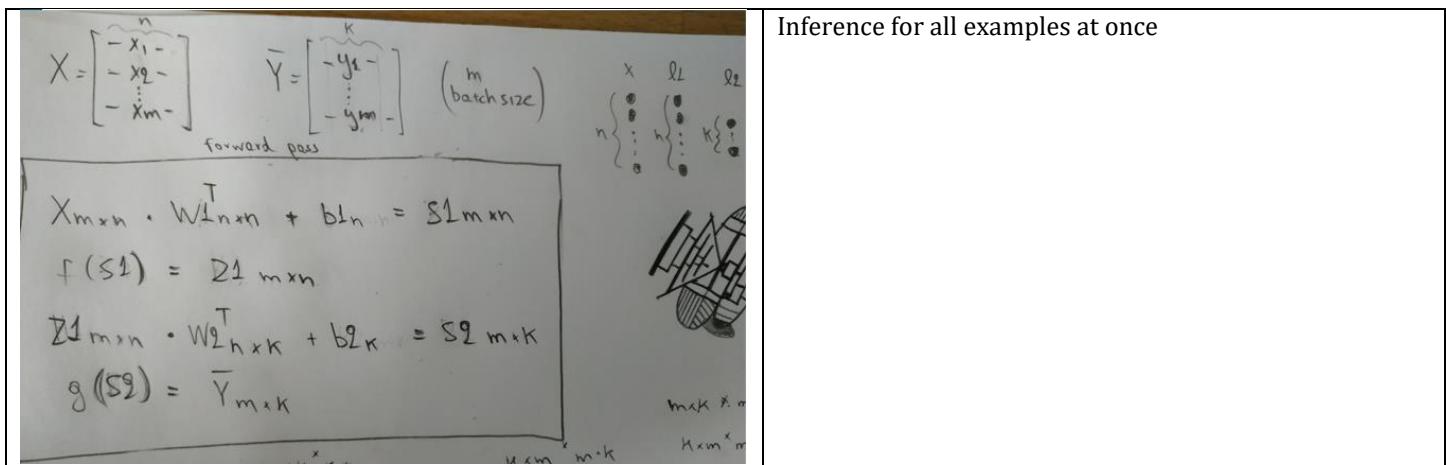
Variable names x observed during: training ✓ — testing ✓ y observed during: training ✓ — testing ✗ z observed during: training ✗ — testing ✗	Bold is vector Curly brackets mean set Curly X is the design matrix (collection of samples)
--	---

The learning rate might be a matrix. (so each parameter (weight) has each one learning rate?)



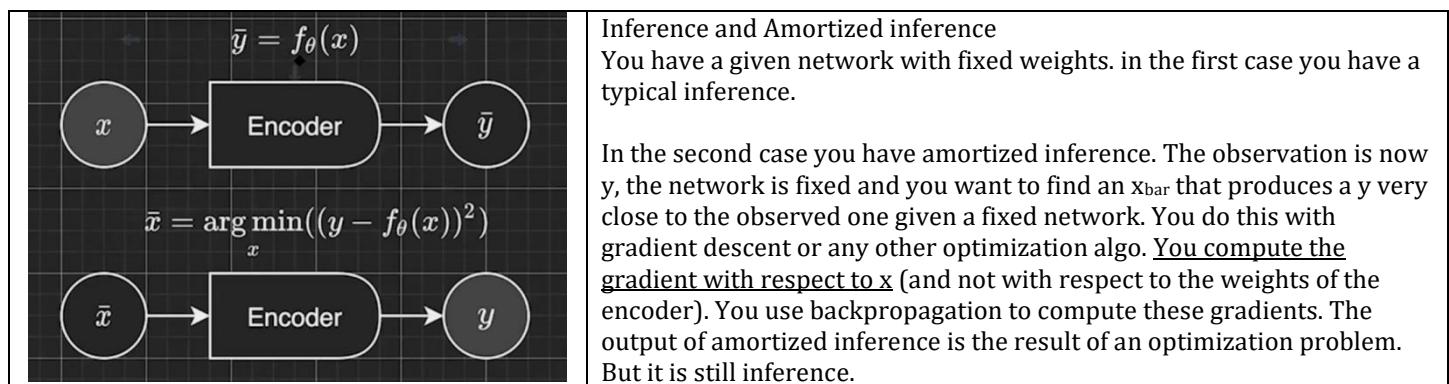
	We use column vectors for the weight matrix. This means that in this case the W matrix between the input (2 units) and the first hidden layer (100 units) would be 100*2 and not 2*100. First is the direction we are looking from. <u>In classification problems transforming to higher dimensions makes the optimization problem easier to solve.</u> The points are separated with each other, there are less constraints and can be much easier to find good linear decision boundaries between them. (<u>Have in mind that this is not a regression problem where you might have sparse data problem by the large number of dimensions</u>)
--	--

<p>Neural network (inference)</p> $\hat{y} \in \mathbb{R}^{K=3} \quad h = f(W_h x + b_h) \quad W_h \in \mathbb{R}^{d \times n}$ $\hat{y} = g(W_y h + b_y) \quad b_h \in \mathbb{R}^d$ $W_y \in \mathbb{R}^{K \times d} \quad b_y \in \mathbb{R}^K$ $f, g = (\cdot)^+, \sigma(\cdot), \tanh(\cdot), \text{soft(argmax)}(\cdot)$ $W_h \in \mathbb{R}^{d=100} \quad \hat{y} = \hat{y}(x), \quad \hat{y} : \mathbb{R}^n \rightarrow \mathbb{R}^K, \quad x \mapsto \hat{y}$ $W_y \in \mathbb{R}^{K \times d} \quad \hat{y} : \mathbb{R}^n \rightarrow \mathbb{R}^d \rightarrow \mathbb{R}^K, \quad d \gg n, K$	<p>C is the correct class index K is the number of classes, let's say 3. First is red, green, blue If the example x_1 is in blue then y_1 encoded with 1-hot encoding will be $[0 \ 0 \ 1]$ and $c=3$, the index (+1) of the class, or the index of the 1 in the y vector. X is the design matrix M examples, n components each example (n features)</p> <p>Variables written in Bold in these diagrams show that they are vectors.</p>
<p>Neural network (training I)</p> <p>logits: output of final layer</p> $\text{soft(argmax}(\mathbf{l})[c] \doteq \frac{\exp(l[c])}{\sum_{k=1}^K \exp(l[k])} \in (0, 1)$ $\mathcal{L}(\hat{Y}, \mathbf{c}) \doteq \frac{1}{m} \sum_{i=1}^m \ell(\hat{y}^{(i)}, \mathbf{c}_i), \quad \ell(\hat{y}, \mathbf{c}) \doteq -\log(\hat{y}[c])$ $\mathbf{x}, \quad c = 1 \quad \Rightarrow \quad \mathbf{y} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$ $\hat{y}(\mathbf{x}) = \begin{pmatrix} \sim 1 \\ \sim 0 \\ \sim 0 \end{pmatrix} \quad \Rightarrow \quad \ell\left(\begin{pmatrix} \sim 1 \\ \sim 0 \\ \sim 0 \end{pmatrix}, 1\right)$	$h = f(W_h x + b_h)$ $\hat{y} = g(W_y h + b_y)$ <p>1- means it tends to 1 from below</p> <p>logit the linear output of a NN is called logit.</p>
<p>Neural network (training II)</p> $\Theta \doteq \{W_h, b_h, W_y, b_y\}$ $J(\Theta) \doteq \mathcal{L}(\hat{Y}(\Theta), \mathbf{c}) \in \mathbb{R}^+$ $\frac{\partial J(\Theta)}{\partial W_y} = \frac{\partial J(\Theta)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial W_y}$ $\frac{\partial J(\Theta)}{\partial W_h} = \frac{\partial J(\Theta)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h} \frac{\partial h}{\partial W_h}$ <p>Train data</p> $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \text{1-hot encoding}$ $\mathbf{X} = \begin{bmatrix} \mathbf{x}^{(1)} \\ \mathbf{x}^{(2)} \\ \vdots \\ \mathbf{x}^{(m)} \end{bmatrix} \quad \mathbf{Y} = \begin{bmatrix} \mathbf{y}^{(1)} \\ \mathbf{y}^{(2)} \\ \vdots \\ \mathbf{y}^{(m)} \end{bmatrix} \quad \mathbf{c} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix}$ $\mathbf{x}^{(i)} \in \mathbb{R}^n \quad \mathbf{y}^{(i)} \in \{0, 1\}^K \quad c_i \in \{1, 2, \dots, K\}$	$h = f(W_h x + b_h)$ $\hat{y} = g(W_y h + b_y)$ <p>gradient descent</p> $\frac{dJ(\vartheta)}{d\vartheta}(\vartheta_0)$ $-\frac{dJ(\vartheta)}{d\vartheta}(\vartheta_0)$ ϑ_0



Inference and Amortized inference

A note: Back propagation is not used only for training. It is used for amortized inference too. gradient descent is used for amortized inference too.



Backpropagation

NG Notes

Backpropagation is a method for calculating the gradient of the cost function with respect to the parameters of the ANN (weights and biases). Then you can use that gradient for an optimization algorithm like gradient descent to minimize the cost function with respect to the parameters of the ANN.

To minimize the cost function of a ANN we can use a gradient based method. For that, we need to be able to calculate the cost function and the partial derivatives of the cost function with respect to its parameters (for many parameter values). Backpropagation is a method using which we can calculate the partial derivatives of the cost function with respect to its parameters and thus we can use them for a gradient based optimization of the cost function (gradient descent or any other gradient based algorithm). Essentially you need a program that gets as input some initial parameters Θ (the initial weights), it calculates the cost function and its partial derivatives for each weight and then it uses these to calculate the new weights by minimizing the cost function.

One example in the training set

Gradient computation

$$\Rightarrow J(\Theta) = -\frac{1}{m} \left[\sum_{i=1}^m \sum_{k=1}^K y_k^{(i)} \log h_\theta(x^{(i)})_k + (1 - y_k^{(i)}) \log(1 - h_\theta(x^{(i)})_k) \right] + \frac{\lambda}{2m} \sum_{l=1}^{L-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} (\Theta_j^{(l)})^2$$

$$\Rightarrow \min_{\Theta} J(\Theta)$$

Need code to compute:

$$\begin{aligned} &\rightarrow -J(\Theta) \\ &\rightarrow -\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) \end{aligned}$$

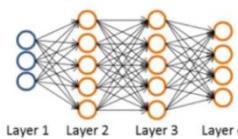
$$\Theta_{ij}^{(l)} \in \mathbb{R}$$

Gradient computation

Given one training example (x, y) :

Forward propagation:

$$\begin{aligned} a^{(1)} &= x \\ z^{(2)} &= \Theta^{(1)} a^{(1)} \\ a^{(2)} &= g(z^{(2)}) \quad (\text{add } a_0^{(2)}) \\ z^{(3)} &= \Theta^{(2)} a^{(2)} \\ a^{(3)} &= g(z^{(3)}) \quad (\text{add } a_0^{(3)}) \\ z^{(4)} &= \Theta^{(3)} a^{(3)} \\ a^{(4)} &= h_\Theta(x) = g(z^{(4)}) \end{aligned}$$



Gradient computation: Backpropagation algorithm

Intuition: $\delta_j^{(l)}$ = "error" of node j in layer l .

$$\begin{aligned} \text{For each output unit (layer } L=4\text{)} \quad &\delta_j^{(4)} = a_j^{(4)} - y_j \quad (\text{h}_\Theta(x)_j) \quad \delta^{(4)} = a^{(4)} - y \\ \rightarrow \delta^{(3)} &= (\Theta^{(3)})^T \delta^{(4)} * g'(z^{(3)}) \\ \rightarrow \delta^{(2)} &= (\Theta^{(2)})^T \delta^{(3)} * g'(z^{(2)}) \\ (\text{No. } \delta^{(1)}) & \end{aligned}$$

If you know the error δ you can calculate the derivative.

$\delta(4)$ is just activation minus training example data. The formula for the other δ turn out to be like the ones shown in the picture where g' is the derivative of the activation function g evaluated at the input value $z^{(3)}$ which is $a^{(3)} * (1 - a^{(3)})$. In general δ is the activation error.

It turns out that in order to calculate the partial derivatives we need first to calculate the error δ for each node. And to do that we need to calculate the δ for the output layer first and use that to calculate the δ of the nodes of the previous layer and so on. So essentially we are back propagating the errors. This is where the name of the method comes from. Of course in order to calculate the δ s we need to calculate the activation of all nodes first and we do this using vectorized forward propagation. For example, in case of one example and with no regularization, the partial derivative of the cost function is given by the formula of the picture $a^* \delta$.

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = a_j^{(l)} \delta_i^{(l+1)}$$

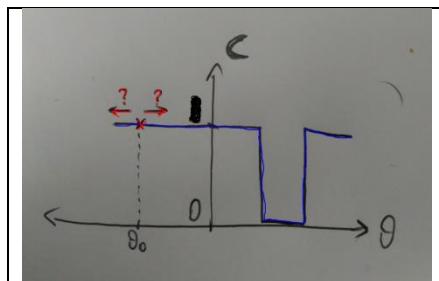
(ignoring λ if $\lambda = 0$)

So if you know the errors, δ , you can calculate the derivative and use it in the minimization calculation of the cost function, for example with the gradient descent method $\rightarrow \theta = \theta - \alpha^* \text{partial-derivative}$ where θ is a weight. You do this for all the weights.

Notice that we use the derivative of the activation function $g'(z)$ in the errors calculation. This means that the activation function g must be differentiable. In mathematics, a differentiable function of one real variable is a function whose derivative exists at each point in its domain. In other words, the graph of a differentiable function has a non-vertical tangent line at each interior point in its domain. A differentiable function is smooth (the function is locally well approximated as a linear function at each interior point) and does not contain any break, angle, or cusp.

* is the element wise multiplication in matlab

The cost function must be smooth. Therefore, ANNs use continuously ranging outputs for the neurons instead of binary outputs as biological neurons do. In the binary output case, a slight change in a weight could be the difference between activation or not, which means that the cost function would be vertical in that position.



Imagine that no matter what the weight is the cost is always the same (the neurons don't fire for example) and for a tiny range of weight values the cost becomes 0. You can't run gradient based optimization on such a cost function. If you begin from a random θ_0 you don't know to which direction to move in order to minimize the cost function.

Many examples in the training set

Backpropagation algorithm

→ Training set $\{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}$

Set $\Delta_{ij}^{(l)} = 0$ (for all l, i, j). (use to compute $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$)

For $i = 1$ to $m \leftarrow (x^{(i)}, y^{(i)})$.

Set $a^{(1)} = x^{(i)}$

Perform forward propagation to compute $a^{(l)}$ for $l = 2, 3, \dots, L$

Using $y^{(i)}$, compute $\delta^{(L)} = a^{(L)} - y^{(i)}$

Compute $\delta^{(L-1)}, \delta^{(L-2)}, \dots, \delta^{(2)}$

$\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$

$\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + \delta_i^{(l+1)} (a^{(l)})^T$

$D_{ij}^{(l)} := \frac{1}{m} \Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)}$ if $j \neq 0$

$D_{ij}^{(l)} := \frac{1}{m} \Delta_{ij}^{(l)}$ if $j = 0$

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)}$$

Now we have many examples so the weight of one node should be calculated based on the error of all examples. The variable Δ accumulates the errors of all examples for each weight. Actually not just the errors but the product $a^* \delta$ (which is the partial derivative of the cost function).

It turns out with some complicated math, that for m examples and regularization, the partial derivative is given by the formula for variable D :

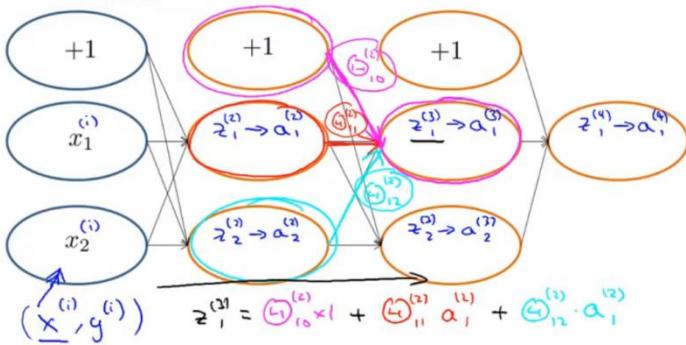
$$D_{ij}^{(l)} := \frac{1}{m} \Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)}$$

$$D_{ij}^{(l)} := \frac{1}{m} \Delta_{ij}^{(l)}$$

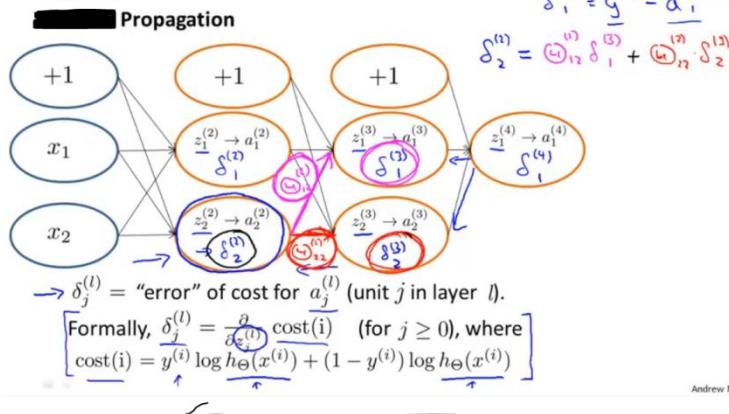
Knowing the derivatives we can use a gradient based method to minimize the cost function for example with the gradient descent method $\rightarrow \theta = \theta - \alpha * \text{partial-derivative}$ where θ is one weight corrected for the errors of all training examples. You do this for all the weights.

Some intuitions for understanding backpropagation

Forward Propagation



Back Propagation



In general, we do a forward and a back propagation calculation for every example of the training set.

So, for one example and if we ignore regularization, forward propagation propagates the activation values forward to calculate the z value of the next node. Back propagation propagates the error values backwards to calculate the error value of the previous node.

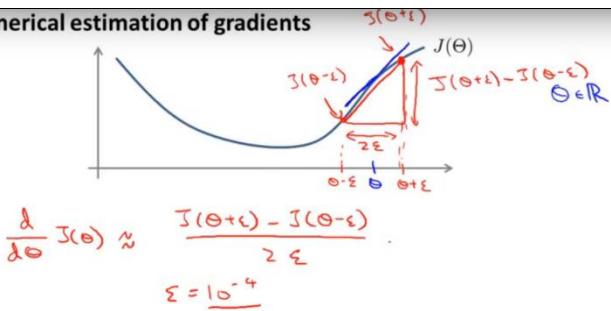
It turns out that the error values are the partial derivatives of the cost function with respect to the z (not with θ).

$\delta_j^{(l)} = \text{"error" of cost for } a_j^{(l)} \text{ (unit } j \text{ in layer } l\text{).}$

Formally, $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(i)$ (for $j \geq 0$), where $\text{cost}(i) = y^{(i)} \log h_\Theta(x^{(i)}) + (1 - y^{(i)}) \log (1 - h_\Theta(x^{(i)}))$

Gradient checking

Numerical estimation of gradients



Parameter vector θ

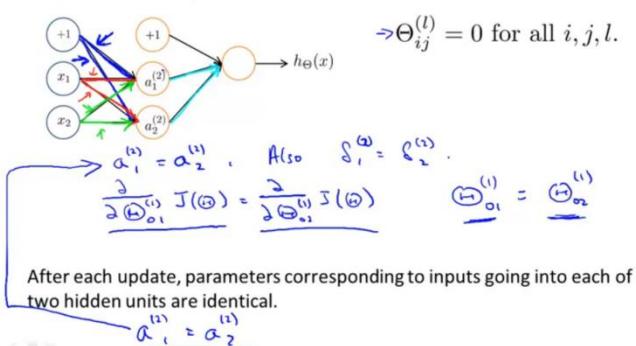
- $\rightarrow \theta \in \mathbb{R}^n$ (E.g. θ is "unrolled" version of $\underline{\theta^{(1)}}, \underline{\theta^{(2)}}, \underline{\theta^{(3)}}$)
- $\rightarrow \theta = [\theta_1, \theta_2, \theta_3, \dots, \theta_n]$
- $\rightarrow \frac{\partial}{\partial \theta_1} J(\theta) \approx \frac{J(\theta_1 + \epsilon, \theta_2, \theta_3, \dots, \theta_n) - J(\theta_1 - \epsilon, \theta_2, \theta_3, \dots, \theta_n)}{2\epsilon}$
- $\rightarrow \frac{\partial}{\partial \theta_2} J(\theta) \approx \frac{J(\theta_1, \theta_2 + \epsilon, \theta_3, \dots, \theta_n) - J(\theta_1, \theta_2 - \epsilon, \theta_3, \dots, \theta_n)}{2\epsilon}$
- \vdots
- $\rightarrow \frac{\partial}{\partial \theta_n} J(\theta) \approx \frac{J(\theta_1, \theta_2, \theta_3, \dots, \theta_n + \epsilon) - J(\theta_1, \theta_2, \theta_3, \dots, \theta_n - \epsilon)}{2\epsilon}$

This is a method with which you can check if the partial gradients computed by back propagation are correct. Essentially it is a way to debug your backpropagation implementation. You calculate numerically the partial derivatives (using the two sided difference method) and compare them with gradient descent. You do this for only one step, one training example and if it is ok then you continue with backpropagation without recomputing numerically the gradients since this computation is very expensive.

Random initialization

To small numbers close to 0, but not 0 since this would raise the problem of symmetric weights. We can't use zero initialization since that would cause all the weights of each node to be identical which means that the nodes of the next layer learn the same function of the input, they have the exact same activation value. Which makes them highly redundant.

Zero initialization



Random initialization: Symmetry breaking

Initialize each $\Theta_{ij}^{(l)}$ to a random value in $[-\epsilon, \epsilon]$ (i.e. $-\epsilon \leq \Theta_{ij}^{(l)} \leq \epsilon$)

E.g.

```
Theta1 = rand(10,11)*(2*INIT_EPSILON)
- INIT_EPSILON;
```

```
Theta2 = rand(1,11)*(2*INIT_EPSILON)
- INIT_EPSILON;
```

Putting it together

Training a neural network

- 1. Randomly initialize weights
 - 2. Implement forward propagation to get $h_\theta(x^{(i)})$ for any $x^{(i)}$
 - 3. Implement code to compute cost function $J(\theta)$
 - 4. Implement backprop to compute partial derivatives $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\theta)$
- for $i = 1:m$ { $(x^{(i)}, y^{(i)})$ $(x^{(i)}, y^{(i)})$, ..., $(x^{(i)}, y^{(i)})$ }
- Perform forward propagation and backpropagation using example $(x^{(i)}, y^{(i)})$ (Get activations $a^{(l)}$ and delta terms $\delta^{(l)}$ for $l = 2, \dots, L$).
- $\Delta^{(2)} := \Delta^{(2)} + \delta^{(2)}(a^{(1)})^\top$
- ...
compute $\frac{\partial}{\partial \Theta_{jk}^{(m)}} J(\theta)$.

Notice that the simplest implementation of back propagation is done with a for loop, but there are more complex vectorized methods that don't use a for loop and are much faster.

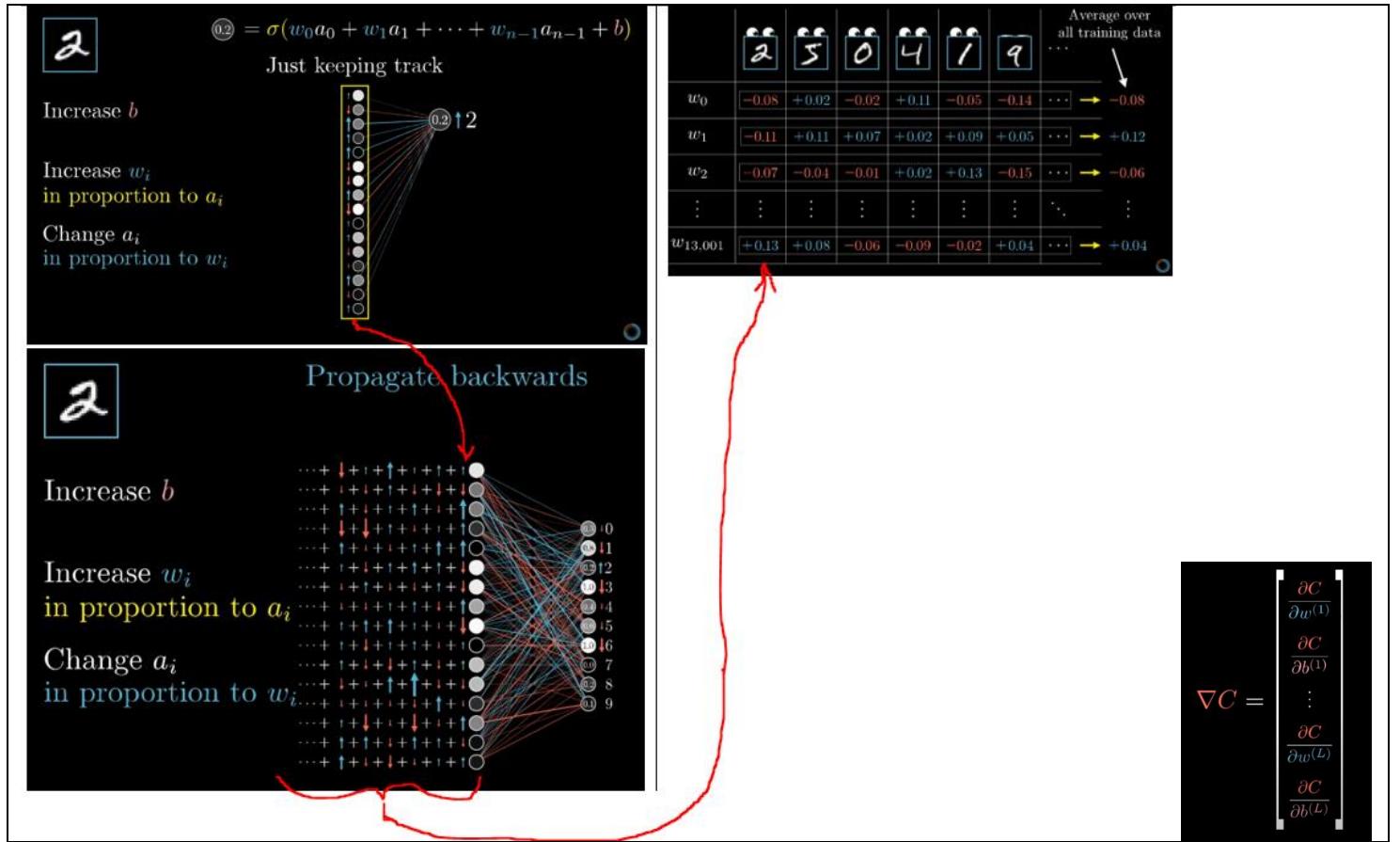
Notice that with this implementation the partial derivative with respect to a parameter (to a weight) is calculated by using information from all training examples. The Δ terms is the accumulated summation of δ s calculated for all examples. So the Δ_{ij} for a specific node j is the cumsum of δ from all training examples. (probably stochastic gradient descent uses only one example)

- 5. Use gradient checking to compare $\frac{\partial}{\partial \Theta_{ik}^{(l)}} J(\Theta)$ computed using backpropagation vs. using numerical estimate of gradient of $J(\Theta)$.
 Then disable gradient checking code.
6. Use gradient descent or advanced optimization method with backpropagation to try to minimize $J(\Theta)$ as a function of parameters Θ

Putting it all together (me)

In a gradient descent step you update all the weights using the calculated partial derivatives from all training examples (the D terms calculated with backpropagation). I guess that then you need to repeat the whole process (steps 1-5) for the updated weights, to calculate the new gradients and use them to re-update the weights. Then for each state with new weights (each epoch) you run the neural net to the test set and you get an error rate (misclassification error) which you add to a plot to watch the algorithm's convergence. In stochastic gradient descent I guess that in each step you calculate the gradients based on some randomly selected examples instead of using all examples in every step.

Intuition



The ANN sees a picture of a 2. It produces an output vector. We know the correct answer so we take the differences between the last layer activation and the correct activation. Each output neuron's activation must be changed by a different factor. Here we isolate the second neuron of the output layer. Its activation needs to be increased by 0.2.

There are three ways to change this activation

- Change bias
- Change weights
- Change previous layer's activation

We can't affect the previous activation but we can affect the bias and weights. So we can calculate how much we should increase each weight (depending on its realted neuron's activation). Increase a lot the weights connected with neurons with large activation etc.

Then we can't affect the activations but we can deduce how we want it to be modified in order to give a better answer for the specific example. Increase the activation of neurons connected with positive weights with the output neuron two, and decrease the ones connected with negative weights. Then we use this result to calculate the weights changes of the previous layer.

This way we get how much we want to change all weights based on this output neuron for thos single example. Then we repeat the same process for the next output neuron, for this specific example. And do this for all output neurons. Then we calculate the average change to each weight given by all output neurons for this specific example. Then repeat this whole process again for each example. And we calculate the average weight change for all examples. These values are actually the negative derivatives of the cost function.

The math

Assume we have an ANN with layers of one neuron (just for understanding the math)

The cost of the (one in this case) output neuron depends on three different terms.

1. The weights of the last layer w^L
2. the bias of the last layer b^L
3. the activations of the previous layer a^{L-1}

The cost sensitivity to the weights (the partial derivative of the cost function with respect to the weight)

$\frac{\partial C_0}{\partial w^{(L)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C_0}{\partial a^{(L)}}$ <p style="text-align: center;">Chain rule</p>	$C_0(\dots) = (a^{(L)} - y)^2$ $z^{(L)} = w^{(L)}a^{(L-1)} + b^{(L)}$ $a^{(L)} = \sigma(z^{(L)})$ <p style="text-align: center;">Desired output</p>	<p>where</p> $\frac{\partial C_0}{\partial a^{(L)}} = 2(a^{(L)} - y)$ $\frac{\partial a^{(L)}}{\partial z^{(L)}} = \sigma'(z^{(L)})$ $\frac{\partial z^{(L)}}{\partial w^{(L)}} = a^{(L-1)}$
$\frac{\partial C_0}{\partial w^{(L)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C_0}{\partial a^{(L)}} = a^{(L-1)}\sigma'(z^{(L)})2(a^{(L)} - y)$		<p>So we know the derivative of the cost function with respect to the last weight w^L for one example.</p> <p>Notice that we use the derivative of the activation function σ and this is the reason for which we want the activation function to be differentiable for back propagation to work.</p>

And for all examples we sum up:

<p>Average of all training examples</p> $\underbrace{\frac{\partial C}{\partial w^{(L)}}}_{\text{Derivative of full cost function}} = \overbrace{\frac{1}{n} \sum_{k=0}^{n-1} \frac{\partial C_k}{\partial w^{(L)}}}^{\text{Derivative of full cost function}}$	<p>And this is one element of the gradient vector. It is the derivative with respect to the last weight w^L. the other elements are the derivatives with respect with the other weights (and biases)</p>
---	---

The cost sensitivity to the bias

$$\frac{\partial C_0}{\partial b^{(L)}} = \frac{\partial z^{(L)}}{\partial b^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C_0}{\partial a^{(L)}} = 1 \sigma'(z^{(L)}) 2(a^{(L)} - y)$$

Again we sum up for all training examples.

The cost sensitivity to the activation of the previous layer

$$\frac{\partial C_0}{\partial a^{(L-1)}} = \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C_0}{\partial a^{(L)}} = w^{(L)} \sigma'(z^{(L)}) 2(a^{(L)} - y)$$

From this part of the cost comes the term back propagation. We have calculated how much we want to modify the previous activation. But we can't directly control it. We can only control indirectly from its own weights, biases and previous activation.
So based on this calculated value, we calculate how we want to change the previous weights and biases and activations.
We actually propagate the error backwards.

$$\nabla C = \begin{bmatrix} \frac{\partial C}{\partial w^{(1)}} \\ \frac{\partial C}{\partial b^{(1)}} \\ \vdots \\ \frac{\partial C}{\partial w^{(L)}} \\ \frac{\partial C}{\partial b^{(L)}} \end{bmatrix}$$

Doing all this, we have calculated the gradient of the cost function. And we can use it with an optimization algorithm.

Compute

Vectorized form for backpropagation

You have a functional module that gets a vector and outputs a vector. This module can be a linear module for example multiplication of the input vector with a matrix, or it can be a nonlinear module for example passing the input through a relu function, or it can be a cost function etc. To back propagate a gradient vector (or scalar, or matrix etc.) through this module, you must first calculate **how each element of its output is affected by each element of its input** (or vice versa however you prefer to think of it). This can be written as the partial derivative of the output vector with respect to the input vector. The result of this operation ($\frac{\partial \text{output}}{\partial \text{input}}$) is a matrix. The first element of the output vector is affected by each element of the input vector which means that this relationship is described by a vector. It shows how much each element of the input affects the first element of the output. You repeat for the second element of the output and so on. So you end up with a matrix with all interactions. (Matrix calculus or multivariate calculus). This is the **jacobian matrix** of that functional module. The jacobian matrix of a functional module shows how each element of the module's output is affected by each element of its input. **So, if you have the gradient vector that you want to backpropagate through that functional module and you know the jacobian of that module with respect to its input** (it will have one jacobian for each set of input parameters), **you can calculate the gradient vector of that input of this module by multiplying the given gradient vector with that jacobian matrix**. **So, you backpropagate the gradient**. Essentially, we know how much each output value should change (this is the gradient that was back propagated to our module from higher parts of the model). We also know how each output node is

affected by each input node. Knowing these two things, we can calculate how much we need to change the nodes of the input to create the given change in the output (this is the gradient that we will back propagate to the lower parts of the model). Each functional module has a certain jacobian that is different for each of its inputs. The jacobian of a functional module which is simply the multiplication of the input by a matrix (a linear functional module), is the transpose of this matrix.

A loss function calculates one scalar value (for example the second norm squared which is a scalar). You want to back propagate that loss. You want to see how this loss value is affected by the input of interest to the cost function module (the y_{hat} is that input). So, you actually need the jacobian of the cost function with respect to y_{hat} . Then you multiply 1 (the derivative of the cost function with respect to itself) with that jacobian and you get the gradient vector of the loss function that you can further propagate.

For more details on various functional modules (activation and cost functions) and their jacobians see

https://aew61.github.io/blog/artificial_neural_networks/1_background/1.b_activation_functions_and_derivatives.html.

https://aew61.github.io/blog/artificial_neural_networks/1_background/1.c_loss_functions_and_derivatives.html

Notice that if you try to do this calculation for all m examples instead of just one and with respect to the parameters W , then you have the derivative of a matrix with respect to a matrix ($dZg/dW2$). The result of this action is a 4d jacobian matrix (think of it as a collection of 3d matrices the same way a 3d matrix can be thought of as a collection of 2d matrices). Again, you see how each element of one matrix is affected by each element of the other one. I'm not sure on how you are supposed to represent this matrix so that the dimensions are ok for multiplications.

Backprop through a functional module

- Using chain rule for vector functions

$$\frac{\partial c}{\partial z_f} = \frac{\partial c}{\partial z_g} \frac{\partial z_g}{\partial z_f}$$

$$[1 \times d_f] = [1 \times d_g] * [d_g \times d_f]$$
- Jacobian matrix
 - Partial derivative of i-th output w.r.t. j-th input

$$\left(\frac{\partial z_g}{\partial z_f} \right)_{ij} = \frac{(\partial z_g)_i}{(\partial z_f)_j}$$

The derivative of the cost C with respect to the vector zf (the activation of the previous layer) is the derivative of the cost with respect to zg (a vector) multiplied by the derivative of zg with respect to zf (a matrix). It is the chain rule which in this case gives a vector matrix multiplication. The matrix is a jacobian matrix. If the g() function is a matrix multiplication, the typical weighted sum with the weights, then the jacobian of that matrix is the transpose version of it. So backpropagating through a linear module just means multiplying gradient with the transpose of the weight matrix.

Have in mind: in general the boxed expression is only true if c is affected by zf only through zg .

Backprop through a functional module

- Using chain rule for vector functions

$$\frac{\partial c}{\partial z_f} = \frac{\partial c}{\partial z_g} \frac{\partial z_g}{\partial z_f}$$

$$[1 \times d_f] = [1 \times d_g] * [d_g \times d_f]$$
- Jacobian matrix
 - Partial derivative of i-th output w.r.t. j-th input

$$\left(\frac{\partial z_g}{\partial z_f} \right)_{ij} = \frac{(\partial z_g)_i}{(\partial z_f)_j}$$

The derivative of the cost with respect to the weights (the bias is included in the weights) is a multiplication of the same vector with a different jacobian matrix. It is the jacobian of the same matrix with respect to the other input so the derivatives are with respect to the weights and not with respect to the activations.

Backprop through a multi-stage graph

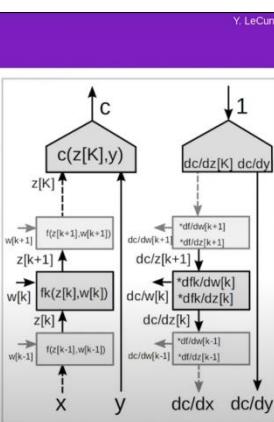
► Using chain rule for vector functions

$$\frac{\partial c}{\partial z_k} = \frac{\partial c}{\partial z_{k+1}} \frac{\partial z_{k+1}}{\partial z_k} = \frac{\partial c}{\partial z_{k+1}} \frac{\partial f_k(z_k, w_k)}{\partial z_k}$$

$$\frac{\partial c}{\partial w_k} = \frac{\partial c}{\partial z_{k+1}} \frac{\partial z_{k+1}}{\partial w_k} = \frac{\partial c}{\partial z_{k+1}} \frac{\partial f_k(z_k, w_k)}{\partial w_k}$$

► Two Jacobian matrices for the module:

- One with respect to $z[k]$
- One with respect to $w[k]$



So with these two formulas you can do backprop.

Gradient, Jacobian,

► Dimensions:

$$y, \bar{y} : [M \times 1] \quad w : [N \times 1]$$

$$\frac{\partial C(y, \bar{y})}{\partial w} = \frac{\partial C(y, \bar{y})}{\partial \bar{y}} \frac{\partial \bar{y}}{\partial w}$$

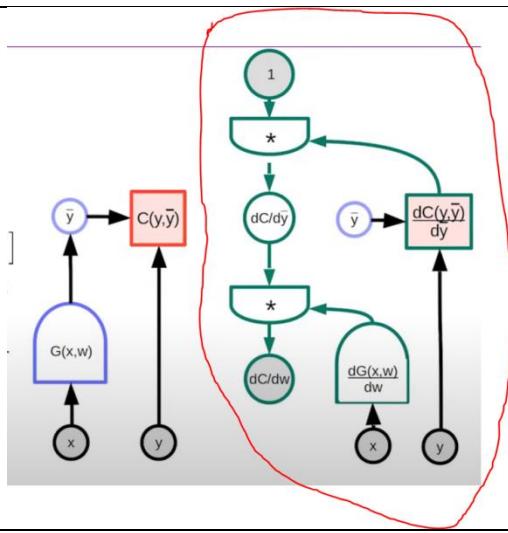
$$[1 \times N] = [1 \times M] \cdot [M \times N]$$

► Row vector = row vector . matrix

$$\frac{\partial C(y, \bar{y})}{\partial w} = \frac{\partial C(y, \bar{y})}{\partial \bar{y}} \frac{\partial G(x, w)}{\partial w}$$

$$[1 \times N] = [1 \times M] \cdot [M \times N]$$

► Gradient = gradient . Jacobian



This typical first graph can be rewritten as the second graph which describes what pytorch and tensorflow do under the hood to achieve automatic differentiation (automatically calculate the derivatives of the cost function using backpropagation). 1 is just the derivative of the cost function with respect to itself. It is just how the loop for backpropagation starts. The * in the graph means multiplication.

Misc

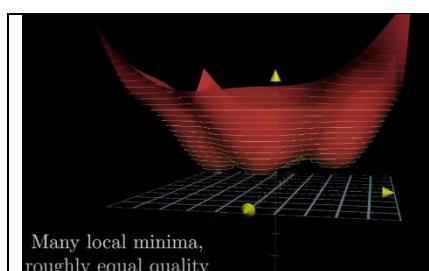
Backpropagation, gradient descent and global optimum

Backpropagation is a method for calculating the cost function. When you have the cost function you can minimize it with respect to your parameters using gradient descent or any other optimization algorithm. Have in mind that the cost function of a NN is a non-convex object so there is no guarantee that gradient based methods will reach to the global optimum. But surprisingly that's ok. Empirically it was found that despite the non-convexity we arrive at sensible solutions. By now there exist some theoretical insights into this:

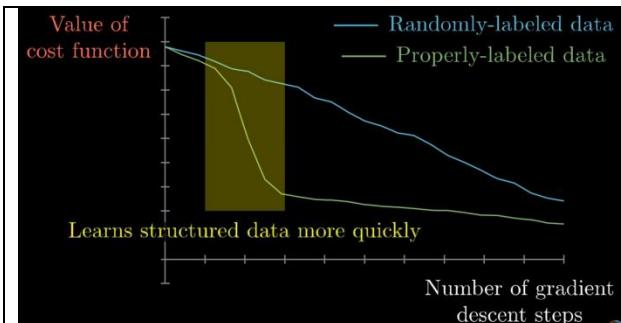
<https://arxiv.org/pdf/1412.0233.pdf>

<https://arxiv.org/pdf/1406.2572.pdf>

The short story is that local minima are rare and they are all very similar to each other and the global minimum.



If the data set has indeed some structure then it turns out that the cost function's local minima are almost of the same quality.



Notice though that if there is no structure in the data, for example in the imangenet case if you change the labels to random words so that for example each image of a lion has a different label, then there is really no structure to your data. Despite this, if the ANN is big enough, it could memorize the entire dataset and produce a good error rate in the training set. The training though would be much slower. If there is structure to the data, ANNs learn faster.

The higher the number of parameters the less of a problem are local minima. Because you might have a local minimum in one parameter but the gradient based algorithm can overcome it by moving to the direction of the other dimension.

The non convex cost function in a NN has many saddle points, areas where the gradient is flat (not going up) in some dimensions. But SGD can usually overcome these regions.

Backpropagation is really one instance of a more general technique called "reverse mode differentiation" to compute derivatives of functions represented in some kind of directed graph form.

Model selection (hyper-parameter tuning)

Hyperparameters: these are parameters that are not being updated by the model (the optimization process on the model). You have to set them and they will be the same during the optimization process. For example in the case of NNs,

- the number of hidden layers
- the number of neurons in each layer
- the learning rate α
- the momentum
- the activation function
- minibatch size
- num of epochs
- dropout

ANNs as space transformations

Block Diagram of a Traditional Neural Net	<small>Y. LeCun</small>
<p>► linear blocks $s_{k+1} = w_k z_k$</p> <p>► Non-linear blocks $z_k = h(s_k)$</p>	<p>ANNs are actually a sequence of linear and non linear blocks. Or linear and nonlinear transformations of the input space. Or in Alfredo terms, a sequence of space rotations and twisting. All these transformations are done with the goal of ending up with a space where the data are linearly separable. You want it to be linearly separable because they can be classified by a linear classifier that a single output unit actually is. For example, in a two class classification you only need one output unit (1 or 0). For more classes each output neuron is still a linear classifier for its class.</p>

(The blocks of an ANN can be represented as a graph that doesn't have loops. You must not have loops for backpropagation to work because you want to know the state of a layer to calculate the state of the previous one).

Mean normalization to data

First you apply mean normalization to the data. The intuition of why: the data are high dimensional vectors (or points if you like). They belong to a very high dimensional space. Let's say they are images of dogs and cats. Notice that the points of these two classes will be very close together, concentrated in a very small region somewhere on the space. The reason for this is that the statistics of these two datasets are extremely similar within a high dimensional space. Cats and dogs are extremely similar objects. You want to zoom on this data cloud to be able to apply decision boundaries on it. to zoom on the data cloud you have to move it to the origin. To do so, must find the mean of this point cloud and subtract it from every data point. This will bring the cloud around 0. Then you want to properly scale it so that it is clearly "visible". To do so, you divide each point with the standard deviation of the cloud. This way, two different datasets with similar distribution but different scales, same clouds but one is larger, (one dataset with points with values around 10k and another dataset with values around 0.1) they all become values around 1. These two steps can zoom any dataset in the origin. Then you start the space transformations to make the data points linearly separable. If you do this mean normalization in a 2d space and your data are around 0 with standard deviation of 1, then the radius of this data cloud would be 3, because in a normal distribution 99% of the data is within 3 standard deviations around the mean.

Rotating data

Whenever you apply a matrix to a vector (matrix vector multiplication) you apply a linear transformation to that vector. In ANNs (in high dimensional space) we have an affine transformation (linear transformation + translation (translation comes from the bias I guess)). This transformation's main component is rotation. Scaling can be put out of the matrix as a scalar. Reflection too (negative scalar is reflection right?). Shear component is small. (If the output layer has fewer units than the input layer, data is rotated and squeezed into a space of fewer dimensions too).

Squeezing data

Alfredo refers to this step as Squashing. How sigmoid or relu twists space? The sigmoid function has a squashing effect. Whatever value x takes, the sigmoid of x is always between 0 and 1. So it squashes (squeezes) the input data. But not twisting it. how? This is how: each vector (meaning each input which can also be represented as a point) passes through a function (which is nonlinear). So it will be moved to some other place after that action. A different vector passing through the same nonlinear function will also be moved but differently. Each vector is moved differently. So the combination of all movements is actually a non linear transformation of the underlying space, and because the points are moved differently (non linearly) shows that the space is actually twisted so for example two points that are far away in a specific region of the initial space, end up close together. And in another region of the initial space, points that are far away end up more far away etc. the space is twisted.

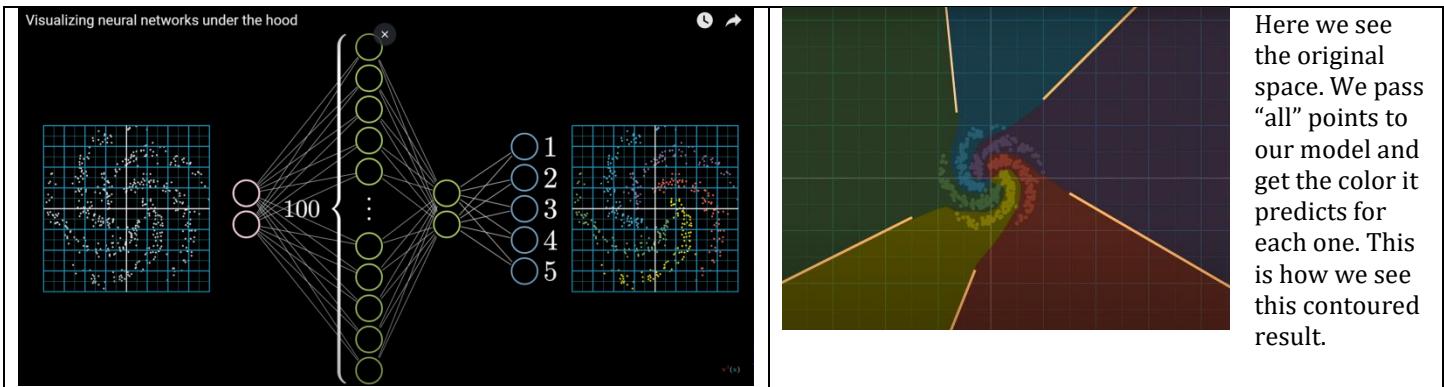
A NN transforms the space in such a way so that the dataset represented in the resulting space of the second to last layer, is linearly separable. Because the last layer is a collection of perceptrons (performing only a weighted sum which is a linear operation), which are linear classifiers.

$$\hat{y} : \mathbb{R}^n \rightarrow \mathbb{R}^d \rightarrow \mathbb{R}^K, \quad d \gg n, K$$

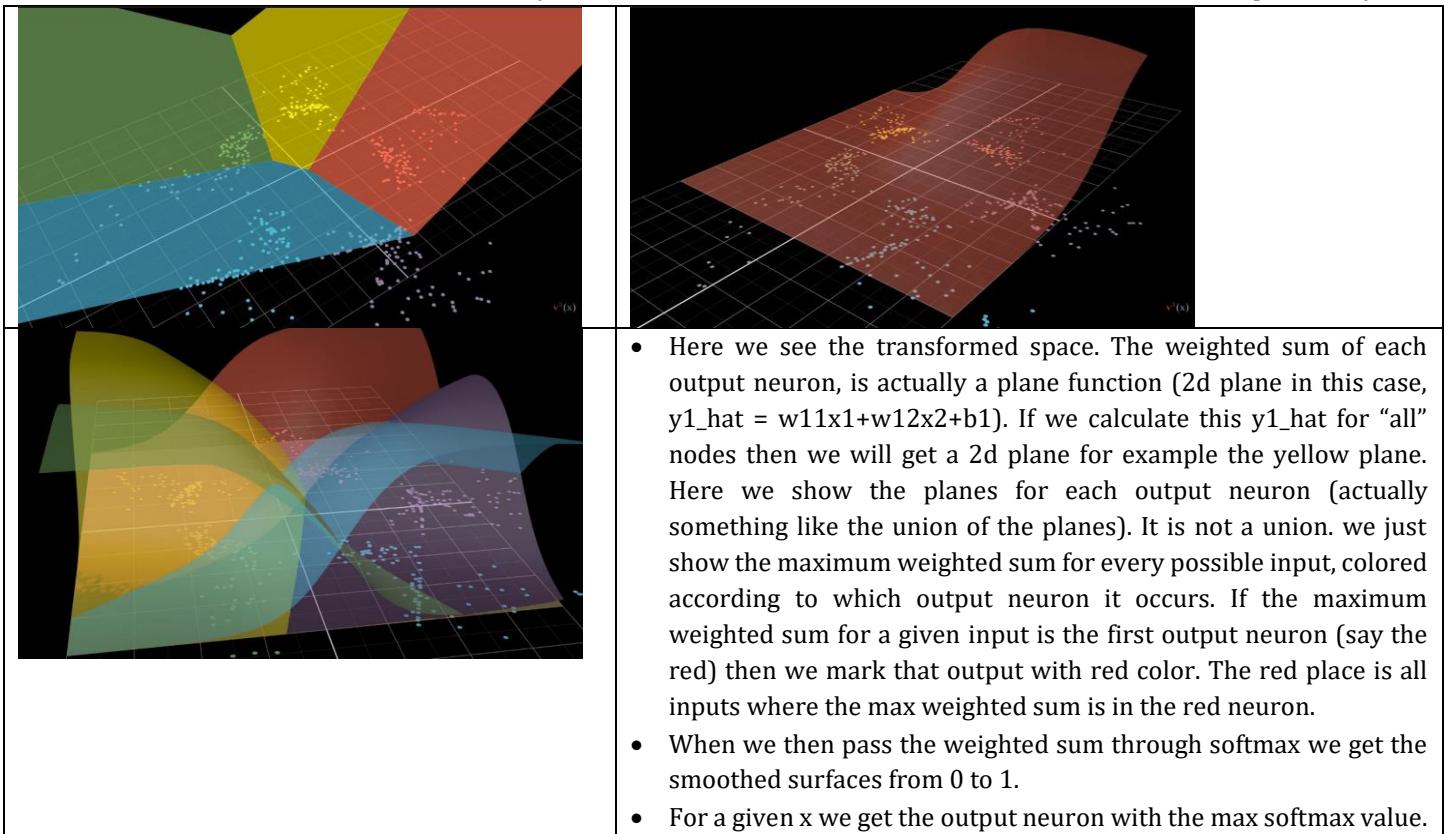
In classification we must first expand the input space to higher dimensions so that the data cloud is separated from the small region of the input space that it usually lies.

Therefore the first hidden layer is always larger than the input layer. In this larger space where the data are more separated with each other the optimization algorithms can work better, transforming the space in a way that the data becomes linearly separable. Then we shrink the space back to the dimensions we want.

The Spiral example



- The second hidden layer which adds one more linear transformation from 100d back to the 2d space, is only added to make it possible to visualize in a 2d space, the result of the transformations made by the first hidden layer. Essentially it allows us to make linear interpolation between the position of a point in the input space and its corresponding position in the transformed space. This way I can visualize how the point moved during the transformation.
- Notice that when we use ReLU for the first hidden layer the decision boundaries out of the data domain are linear. This is not extrapolation, it is just linear expansion of the boundaries. Extrapolation would be to continue the spiral shape outwards. If we used a sinusoidal activation function they would be kind of sinusoidal and would contain some short of periodicity.



Pytorch 5 basic steps

```

learning_rate = 1e-3
lambda_l2 = 1e-5

# nn package to create our Linear model
# each Linear module has a weight and bias
model = nn.Sequential(
    nn.Linear(D, H),
    nn.Linear(H, C)
)
model.to(device) #Convert to CUDA

# nn package also has different loss functions.
# we use cross entropy loss for our classification task
criterion = torch.nn.CrossEntropyLoss()

# we use the optim package to apply
# stochastic gradient descent for our parameter updates
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate, weight_decay=lambda_l2) # built-in L2

# Training
for t in range(1000):

    # Feed forward to get the Logits
    1 → y_pred = model(X)

    # Compute the Loss and accuracy
    2 → loss = criterion(y_pred, y)
    score, predicted = torch.max(y_pred, 1)
    acc = (y == predicted).sum().float() / len(y)
    print("[EPOCH]: %d, [LOSS]: %.6f, [ACCURACY]: %.3f" % (t, loss.item(), acc))
    display.clear_output(wait=True)

    # zero the gradients before running
    # the backward pass.
    3 → optimizer.zero_grad()

    # Backward pass to compute the gradient
    # of Loss w.r.t our Learnable params.
    4 → loss.backward()

    # Update params.
    5 → optimizer.step()

```

Training for classification in Pytorch is composed of 5 basic steps:

1. Get the prediction with a Forward pass
2. Calculate the prediction loss
3. Make the gradients 0
4. Calculate the gradients with respect to the parameters
5. Move to the direction of maximum loss decrease (opposite of the direction of the gradient of the cost function)

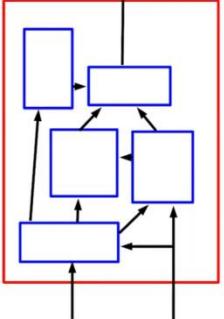
In pytorch gradients are accumulated. This is convenient for various reasons for complicated architectures. The `backwards()` function accumulates the gradients. In this case, we don't want that. Therefore at each optimization step (each epoch) we make the gradients 0 before accumulating.

`crossEntropyLoss()` is the negative logsoftmax

`optim.SGD` (stochastic gradient descent)
`optim.Adam` (a similar optimization algo)

Architecture

We already know about the linear modules. It's an (input) vector (weight) matrix multiplication.

Basic Modules	Basic modules with their jacobians
Linear $Y = W.X$; $dC/dX = W^T \cdot dC/dY$; $dC/dW = X \cdot dC/dY$ ReLU $y = \text{ReLU}(x)$; if ($x < 0$) $dC/dx = 0$ else $dC/dx = dC/dy$ Duplicate $Y_1 = X, Y_2 = X$; $dC/dX = dC/dY_1 + dC/dY_2$ Add $Y = X_1 + X_2$; $dC/dX_1 = dC/dY$; $dC/dX_2 = dC/dY$ Max $y = \max(x_1, x_2)$; if ($x_1 > x_2$) $dC/dx_1 = dC/dy$ else $dC/dx_1 = 0$ LogSoftMax $Y_i = X_i - \log[\sum_j \exp(X_j)]$;???	
Any directed acyclic graph is OK for backprop	
<ul style="list-style-type: none"> ▶ As long as there exist a partial order on the modules ▶ If the graph has loops, we need to "unroll" them. ▶ Recurrent networks and backprop through time 	

Activation functions

Nonlinear activation functions

- Point wise nonlinearities

They transform nonlinearly one point to another. The weighted sum to the output of the activation function. A single value to a single value. For example, **ReLU**, sigmoid etc.

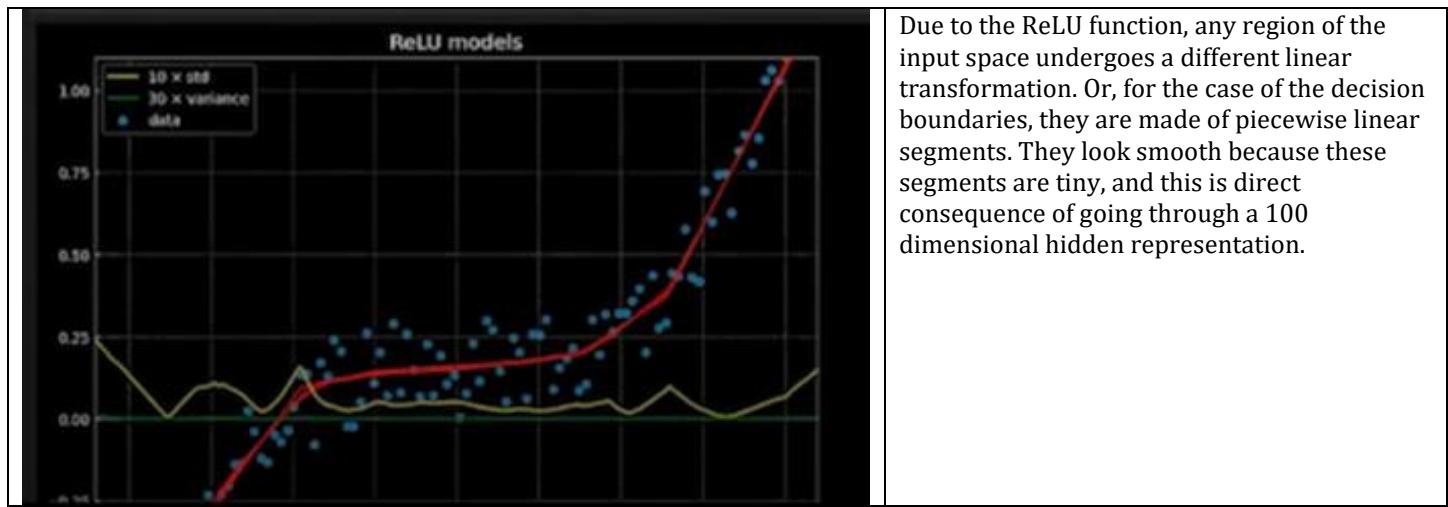
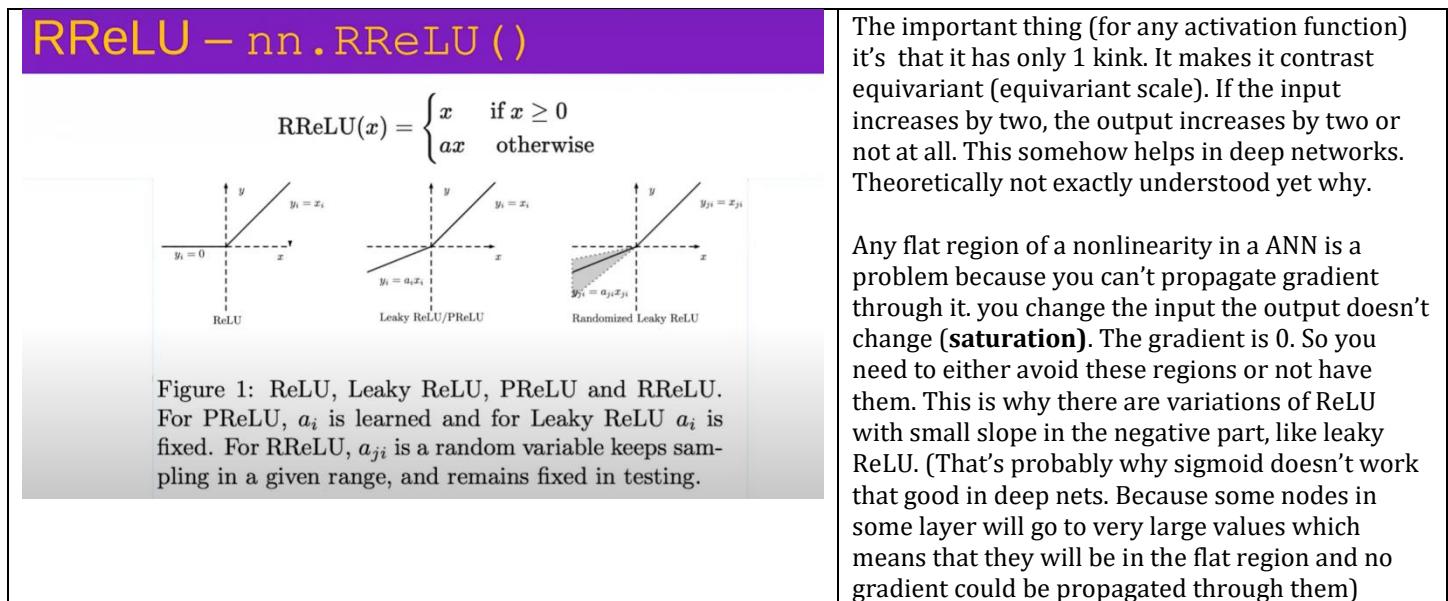
- Vector to vector nonlinearities

They transform nonlinearly the entire vector to another one. A function is applied to all weighted sums and changes each value of the vector but in a way that is related with the changes to the other values of the vector. It is applied to the vector as a whole and outputs a new vector of the same size. For example, **softmax**.

➤ Point wise nonlinearities

ReLUs

ReLU (rectified linear unit) is called the positive part function in math. $\max(0, x)$ in latex. It turned out that using ReLU makes the training process much more efficient.



Sigmoid

Go to function to use if you want to make a decision between a or b, for example, if you want to activate a part of the network or not, a sigmoid is a good function to use to compute the coefficient that activates/deactivates (in a differentially manner).

Tanh

Softshrink

Used in sparse coding

Logsigmoid

General tip

Don't use non monotonic activation functions because two different inputs have the same output and that doesn't do good to gradient descent. (It can lead to increase in the number of local minima or it can introduce saddle points)

➤ Vector to vector non-linearities

Softmax

and softmin

nn.Softmax()

Applies the Softmax function to an n-dimensional input Tensor rescaling them so that the elements of the n-dimensional output Tensor lie in the range [0,1] and sum to 1.

Softmax is defined as:

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

They should have been called argsoftmax and argsoftmin. Argmax get the maximum value of a vector and makes all other values 0. Argsoftmax doesn't make all others 0 but they go to 0 smoothly, so the second largest would be non zero etc. they are all summed up to 1. So softmax can turn the arbitrary scores output from a layer to a probability distribution of discrete variable.

$$\text{Softmax}(x + c) = \text{softmax}(x)$$

If you add the same constant to all elements of the input vector the softmax doesn't change. This means that softmax doesn't care about the amplitudes of the data it only cares about their relative size difference. This is very useful.

$$\text{Softmax}(b*x)$$

By varying the scalar b, you vary the smoothness of the softmax. If b is small softmax is smooth. If it is large, let's say 1000, then it is not smooth at all. Essentially the biggest value will be 1 and all others 0 or so.

Just have in mind:

The argmax function is the derivative of the max

The softargmax is the derivative of the argmax.

Important

Nomenclature and PyTorch

$$\underset{actual\text{-softmax}}{\text{softmax}_{\beta}[E(\mathbf{y}, z)]} \doteq \frac{1}{\beta} \log \sum_{z \in \mathcal{Z}} \exp[\beta E(\mathbf{y}, z)] - \frac{1}{\beta} \log N_z$$

$\frac{|\mathcal{Z}|}{\Delta z}$

$$= \frac{1}{\beta} \text{torch.logsumexp}(\beta E(\mathbf{y}, z), \text{dim}=z)$$

$$\underset{actual\text{-softmax}}{\text{softmin}_{\beta}[E(\mathbf{y}, z)]} \doteq -\frac{1}{\beta} \log \frac{1}{N_z} \sum_{z \in \mathcal{Z}} \exp[-\beta E(\mathbf{y}, z)]$$

$$= \text{softmax}_{\beta}[-E(\mathbf{y}, z)]$$

$$\text{torch.softmax}(l(j), \text{dim}=j) = \text{softargmax}_{\beta=1}[l(j)]_j$$

In pytorch the names of the functions are wrong as of 2021!

`torch.softmax`
in reality it is the softargmax

the actual softmax is:
 $1/b * \text{torch.logsumexp}(b * E)$

The actual softmin is:
`-actual_softmax(-E)`

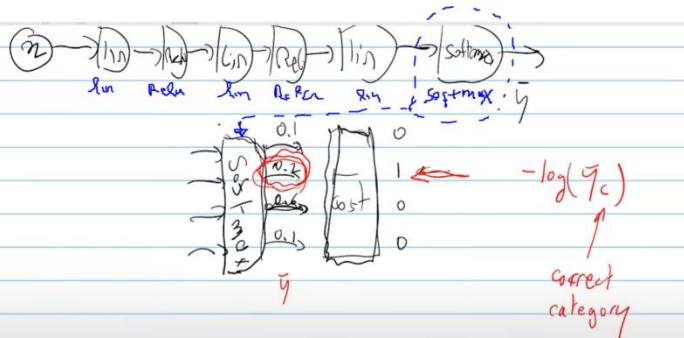
Cost functions

logsoftmax

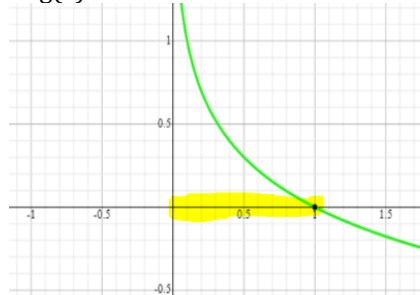
The most used cost function for classification problems

`nn.LogSoftmax()`

$$\text{LogSoftmax}(x_i) = \log \left(\frac{\exp(x_i)}{\sum_j \exp(x_j)} \right)$$



`-log(x)`



In classification problems, after the softmax of the last layer there is a very important module. The cost function module. Or loss function. There are various ways to calculate a cost function for your problem. One way is the square error (the sum of the squares of the difference between the predicted output and the correct output). Another way is the negative logarithm of the predicted value of the correct class $-\log(y_c)$.

The $-\log(x)$ function is 0 if $x=1$ and infinity if $x=0$. So if the output of the correct class is close to 1 the error calculated this way would be close to 0 and vice versa. And since the output values of the units are competitive (softmax), it means that if we increase the correct one the others must be decreased. **So the cost function (the objective function) is to minimize the negative logarithm of the logsoftmax function of the correct class.** (or in other words, the negative log likelihood of the correct class)

In general, **you should calculate the logsoftmax in one go**, and not first the softmax values and then the log of it. the reason is to avoid numerical issues in the computations, issues that arise due to large numbers (for example $\log(0)=-\infty$). Pytorch does a trick in logsoftmax code, to avoid those numerical problems. **So you actually use the logsoftmax at the last layer directly instead of the softmax.** You compute the cost directly without computing the predicted values. You only need them to calculate the loss after all.

CE – nn.CrossEntropyLoss()

Y. LeCun

The loss can be described as:

$$\text{loss}(x, \text{class}) = -\log \left(\frac{\exp(x[\text{class}])}{\sum_j \exp(x[j])} \right) = -x[\text{class}] + \log \left(\sum_j \exp(x[j]) \right)$$

or in the case of the `weight` argument being specified:

$$\text{loss}(x, \text{class}) = \text{weight}[\text{class}] \left(-x[\text{class}] + \log \left(\sum_j \exp(x[j]) \right) \right)$$

The losses are averaged across observations for each minibatch.

Can also be used for higher dimension inputs, such as 2D images, by providing an input of size $(\text{minibatch}, C, d_1, d_2, \dots, d_K)$ with $K \geq 1$, where K is the number of dimensions, and a target of appropriate shape (see below).

Logsoftmax is a special case of cross entropy error category.

Have in mind this intuition about the loss produced by the logsoftmax. The ratio of logs equals the difference of these logs. The log with the exponential cancels out so you are left with the right part of the equation as shown in the picture (first row). This says that to minimize this expression you must increase the weighted sum of the correct class (the $x[\text{class}]$) and decrease the sum of the exponentials of the other weighted sums (the log partition function) including the correct one.

This ends up in increasing the correct weighted sum and decreasing the others.

$$J_{\text{softmax}} = \begin{pmatrix} \frac{\partial s_1}{\partial z_1} & \frac{\partial s_1}{\partial z_2} & \dots & \frac{\partial s_1}{\partial z_n} \\ \frac{\partial s_2}{\partial z_1} & \frac{\partial s_2}{\partial z_2} & \dots & \frac{\partial s_2}{\partial z_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial s_n}{\partial z_1} & \frac{\partial s_n}{\partial z_2} & \dots & \frac{\partial s_n}{\partial z_n} \end{pmatrix}$$

We know the cost vector. To start the back propagation loop, we can calculate the gradient of the cost with respect to itself (c/dc which is 1). (But if it is 1 how the magnitude of cost value affects the gradient descent algorithm? I think it doesn't. we only care about the gradient of the cost function w.r.t the parameters. The value of the cost itself is not indicative for anything, for example the minimum value can be very large.) Then we want to back propagate this first gradient through the cost module. The cost module is a function. The logsoftmax in this case. So we need to calculate the jacobian of the logsoftmax function (how each input affects each output). Then we can multiply the cost gradient value with respect to itself with the jacobian of the logsoftmax to get the gradient of the cost with respect to the input to the cost function.

As I understand it, using softmax (and so logsoftmax as the cost function) there is one gradient propagated back for each example. While in the mean square error for example, each output neuron contributes a distinct gradient and their contributions are added. Is this correct? No. all cost functions output a scalar value which is the total cost of the current input. In order to find the gradient vector to back propagate in the other modules, you need the jacobian of the cost function with which you multiply by 1. If the input to the cost function is a vector and the output a scalar, the jacobian is a vector. Multiplied by a scalar gives the gradient vector.

$$z_i = \frac{e^{\beta z_i}}{\sum_j e^{\beta z_j}}$$

$\frac{\partial c}{\partial z_i}$ known for all i

compute $\frac{\partial c}{\partial x_k} = \sum_i \frac{\partial z_i}{\partial x_k} \cdot \frac{\partial c}{\partial z_i}$

\uparrow Jacobian matrix of softmax

Backpropagation through a softmax. Finding this jacobian is a good exercise.

nn.CrossEntropyLoss()

CrossEntropyLoss is mainly used for multi-class classification

it takes a Y which is not one hot encoded (the first label would be $Y=[0]$, the second label $Y=[1]$ etc.)

it takes a `y_pred` which contains the logits (not the softmax of them because crossentropy loss calculates the softmax and then the negative log of it internally)

nn.BCELoss()

mainly used for binary classification

have in mind the binary cross entropy loss for binary classification problems (two classes 0 or 1)

bceloss vs crossentropyloss

When CrossEntropyLoss is used for binary classification, it expects 2 output features. Eg. `logits=[-2.34, 3.45]`, `Argmax(logits) → class 1`

When BCEloss is used for binary classification, it expects 1 output feature. Eg 1. `logits=[-2.34]<0 → class 0`; Eg 2. `logits=[3.45]>0→class 1`

For CrossEntropyLoss, softmax is a more suitable method for getting probability output. However, for binary classification when there are only 2 values, the output from softmax is always going to be something like [0.1%, 99.9%] or [99.9%, 0.1%] based on its formula. Eg. `softmax([-2.34, 3.45])=[0.3%, 99.7%]`. It does not represent meaningful probability. So softmax is only suitable for multi-class classification.

```
# Binary classification
class NeuralNet1(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(NeuralNet1, self).__init__()
        self.linear1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.linear2 = nn.Linear(hidden_size, 1)

    def forward(self, x):
        out = self.linear1(x)
        out = self.relu(out)
        out = self.linear2(out)
        # sigmoid at the end
        y_pred = torch.sigmoid(out)
        return y_pred

model = NeuralNet1(input_size=28*28, hidden_size=5)
criterion = nn.BCELoss()
```

nn.NLLLoss

Negative loss likelihood

The cross-entropy loss and the (negative) log-likelihood are the same in the following sense: If you apply Pytorch's CrossEntropyLoss to your output layer, you get the same result as applying Pytorch's NLLLoss to a LogSoftmax layer added after your original output layer.

```

class FC2Layer(nn.Module):
    def __init__(self, input_size, n_hidden, output_size):
        super(FC2Layer, self).__init__()
        self.input_size = input_size
        self.network = nn.Sequential(
            nn.Linear(input_size, n_hidden),
            nn.ReLU(),
            nn.Linear(n_hidden, n_hidden),
            nn.ReLU(),
            nn.Linear(n_hidden, output_size),
            nn.LogSoftmax(dim=1)
        )

    def forward(self, x):
        x = x.view(-1, self.input_size)
        return self.network(x)

```

```

optimizer.zero_grad()
output = model(data)
loss = F.nll_loss(output, target)
loss.backward()
optimizer.step()

```

Mean square error

It is good for regression problems (continuous output instead of discrete output like classification). You might see it as

$1/n * ||\mathbf{y} - \hat{\mathbf{y}}||^2$ (the L2 norm) instead of sum of individual differences. (The L1 norm is the sum of the absolute values of the difference instead of the squares of the difference.). the L2 norm is the Euclidean distance of the vector. We need to square it so that the value shows the mean square error and not the square root of the mean square error.

NLL (Negative logarithm likelihood) loss

Logsoftmax is a special case of this category. If the output has come out of a softmax, then this method is actually the logsoftmax.

Adaptivelogsoftmax

It used in cases in which you have extremely large number of classification categories. For example, in language models where you have to predict the next possible word and the ANN gives a probability to all words of the vocabulary. This method is actually neglecting the low probability classes.

marginRankingloss

You ensure that the correctly predicted output will have at least a margin (distance) with the next most probable one.

Multiplicative modules (attention)

Multiplicative Modules

► Quadratic layer, product units, Sigma-Pi units

$$s_i = \sum_j w_{ij} x_j \text{ with : } w_{ij} = \sum_k u_{ijk} z_k ; \text{ equiv : } s_i = \sum_{jk} u_{ijk} z_k x_j$$

► Attention module

$$s_i = \sum_j w_j x_{ij} \quad w_j = \frac{e^{z_j}}{\sum_k e^{z_k}}$$

In these modules the weights w are the result of another ANN with weights u . This way, z can activate or deactivate certain parts of the Wx network. Specific z can make some weights 0 and others 1, essentially deactivating/activating parts of the network. This effect was named attention which is probably a bad name. The reason is that z makes the system to pay attention in some parts of its network and ignore others.

It is called quadratic because you end up multiplying z and x to calculate s , so if x and z are the same thing you get a quadratic formula (a square).

This module is used a lot and can implement switches.

Soft switch

Assume that the system is fed with one common input, $z=x_1=x_2$. The u module is a softmax. In the simple case where w_1 and w_2 are scalar values and not matrices, softmax receives an input vector of 2 dimensions (2 values) and outputs 2 values between 0 and 1 which sum up to 1. These two values are the weights of the subsequent modules, w_1 and w_2 . If it produces 1 and 0 then it selects the first part of the network and deactivates the other completely. If softmax gives intermediate values then you get a linear combination of the two networks. So it is a soft switch between the two networks. Notice that softmax learns to decide to which part of the system to forward a certain input. Or in other words learn to decide to which part of the input space each subsequent network is an expert to. This mechanism is used to build mixture of experts systems. The large google language model with billions of parameters is such an implementation.

Mixture of Experts

► Attention module “Switches” expert networks

$$s_i = \sum_j w_j x_{ij} \quad w_j = \frac{e^{z_j}}{\sum_k e^{z_k}}$$

The Gater learns to decide to which part of the space each expert is an expert on.

Assume you build a speech recognition system. It should be able to process more than one languages that are similar to each other. A new sound is received. Each expert can process one language. Gater decides to which language the sound belongs and forwards it to the corresponding expert.

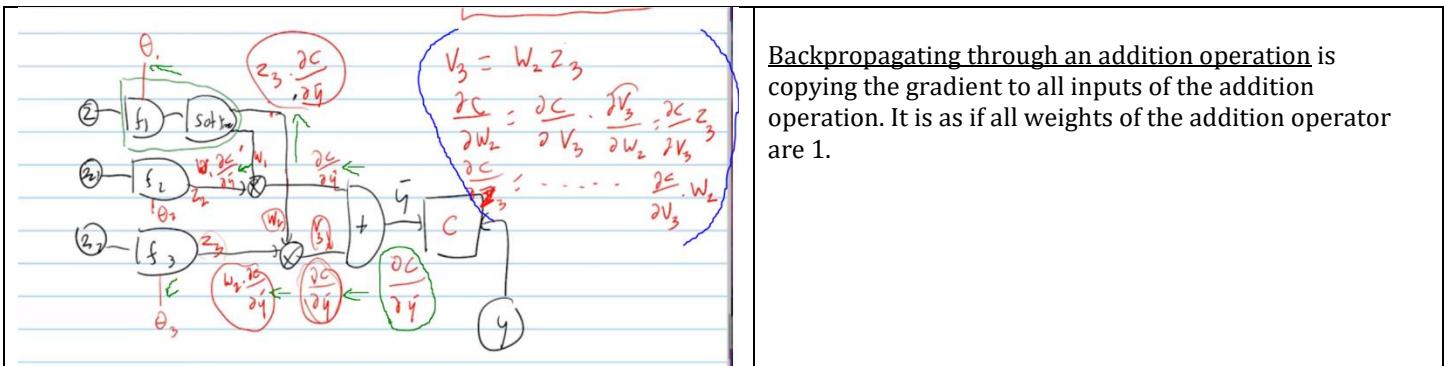
If you simplify it in 2 dimensions. Your data aren't linearly separable. But you can eventually do this nonlinear classification with 3 linear classifiers in parallel one of which decides which expert to use. See the picture.

This is a module in pytorch. You can just add it. it does automatic differentiation to compute the gradients and perform backpropagation.

Notice that softmax in this case outputs a 2d vector while the weight matrices are scalars. In the general case you would have higher dimensions. And you get a product of x_1 (and x_2 etc.) with a slice of the softmax output (a slice of this tensor)

Some notes

- A single layer ANN is a linear classifier
- The “arg” prefix in general means that instead of using the value we use the index of the value.



Backpropagating through an addition operation is copying the gradient to all inputs of the addition operation. It is as if all weights of the addition operator are 1.

Hard switch

You can implement a hard switch too (without any linear interpolation between networks, it's one or the other).

Parameter transformations

Y. LeCun

► When the parameter vector is the output of a function

$$w = H(u)$$

$$u \leftarrow u - \eta \frac{\partial H^T}{\partial u} \frac{\partial C^T}{\partial w}$$

$$w \leftarrow w - \eta \frac{\partial H}{\partial u} \frac{\partial H^T}{\partial u} \frac{\partial C^T}{\partial w}$$

$$[N_w \times N_u] [N_u \times N_w] [N_w \times 1]$$

$$\frac{\partial C}{\partial u} = \frac{\partial C}{\partial z_1} + \frac{\partial C}{\partial z_2}$$

$$\frac{\partial C}{\partial w} = \frac{\partial C}{\partial z_1} \frac{\partial z_1}{\partial w} + \frac{\partial C}{\partial z_2} \frac{\partial z_2}{\partial w}$$

$$\frac{\partial C}{\partial w} = \frac{\partial C}{\partial z_1} + \frac{\partial C}{\partial z_2}$$

A generic case where the weights of the $G(x,w)$ module are themselves a function $H(u)$ of another input u . You minimize the cost by tuning u . This tunes indirectly the weights w . (These networks are also called hypernetworks)

The second picture, is a special case of this architecture where $H(u)$ is just a copy. The parameter u (assume it is scalar here) is copied multiple times and creates a w vector whose values are all u .

Backpropagating through a module that copies its input (let's say its scalar input) to a vector where each value of the vector is a copy of the input, is adding the gradients of all copies.

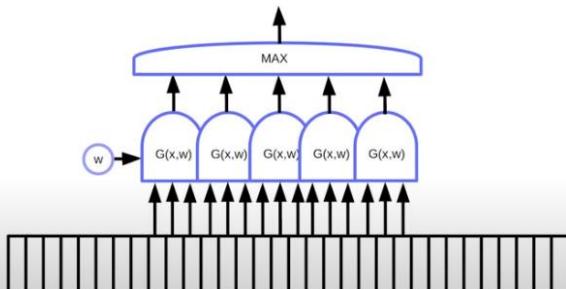
And vice versa, back propagation through a module that adds, is copying the gradient to all components of the addition.

This architecture is used a lot, and it is the basis for a technique called shared weights, used in convolutional nets where you want to identify motifs in any position of the input (motif in an image, in a timeseries etc.). You copy the same module in all positions of the input. The weights of the copied module are w .

You can map this to the previous simple example where the scalar parameter u is copied to create the

Shared Weights for Motif Detection

Detecting motifs anywhere on an input



vector w and the gradient of u is the sum of the w gradients. Here the W matrix is the u parameter, which is copied multiple times to form a higher order matrix with its copies. Then the gradient of w is the sum of the gradients of each module's w .

So this way you can copy (or slide) a module to the whole input and train the weights of this basis module by adding the gradients of all positions of the module.

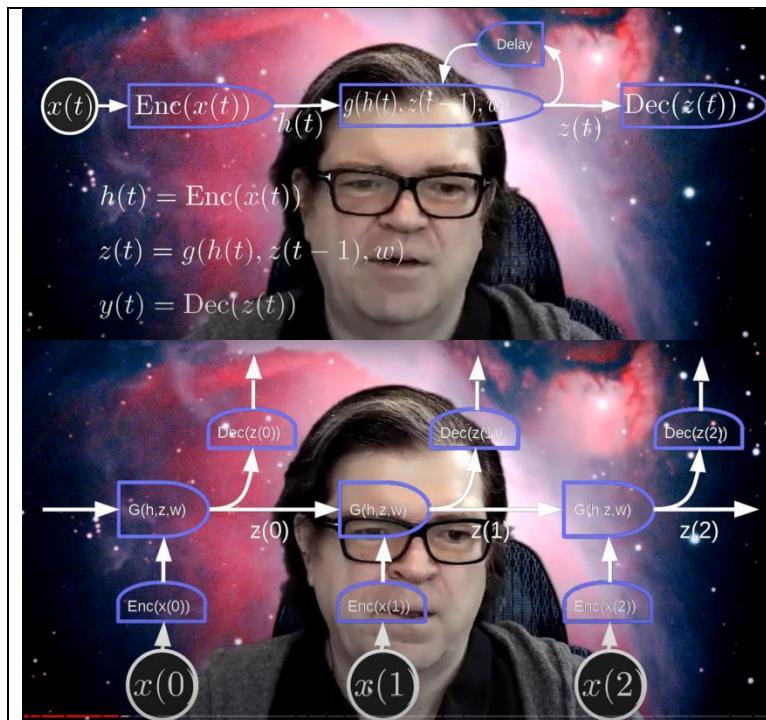
Essentially you can think of it as a big module assembled by a basis module and the weights of this basis module (of this region of the assembled module) are shared (copied) with other regions of the assembled module. The parameters that need to be tuned in this case are the weights w of the submodule that assembles the big module.

During inference you just have to slide the component over the input. During training: Backpropagating through a max function propagates gradient only to the maximum component. The other components tiny perturbation (dx) doesn't affect the output. So, during training a system with shared weights with a max function only changes the parameters of the component that has the maximum output. The gradients of the other modules are 0. So they don't contribute to the summation of gradients that calculates the total parameters gradient. But if the operation is a softmax then the other components gradient contributes to the total gradient.

The gradients are summed (or accumulated). This is the reason why PyTorch accumulates gradients by default. To make it easy to work with this very common architecture used in convolutional and recurrent artificial neural networks.

RNNs

In a nutshell



The delay module acts as a memory. It stores the current produced state $z(t)$, to reuse it as input to the g module in the next step. Z is a recursive function (it calls itself). Time is discretized. This architecture allows the NN to have a memory. It is useful for processing sequences. Both time or spatial sequences. The input is a sequence, $x(0), x(1), x(2)$ etc. It is a cyclic graph, but the trick is that **it is unfolded in time** so it becomes acyclic. The $z(t)$ state goes to the g function of the next step not of the same step. The encoder and decoder have trained parameters too that are also shared across time.

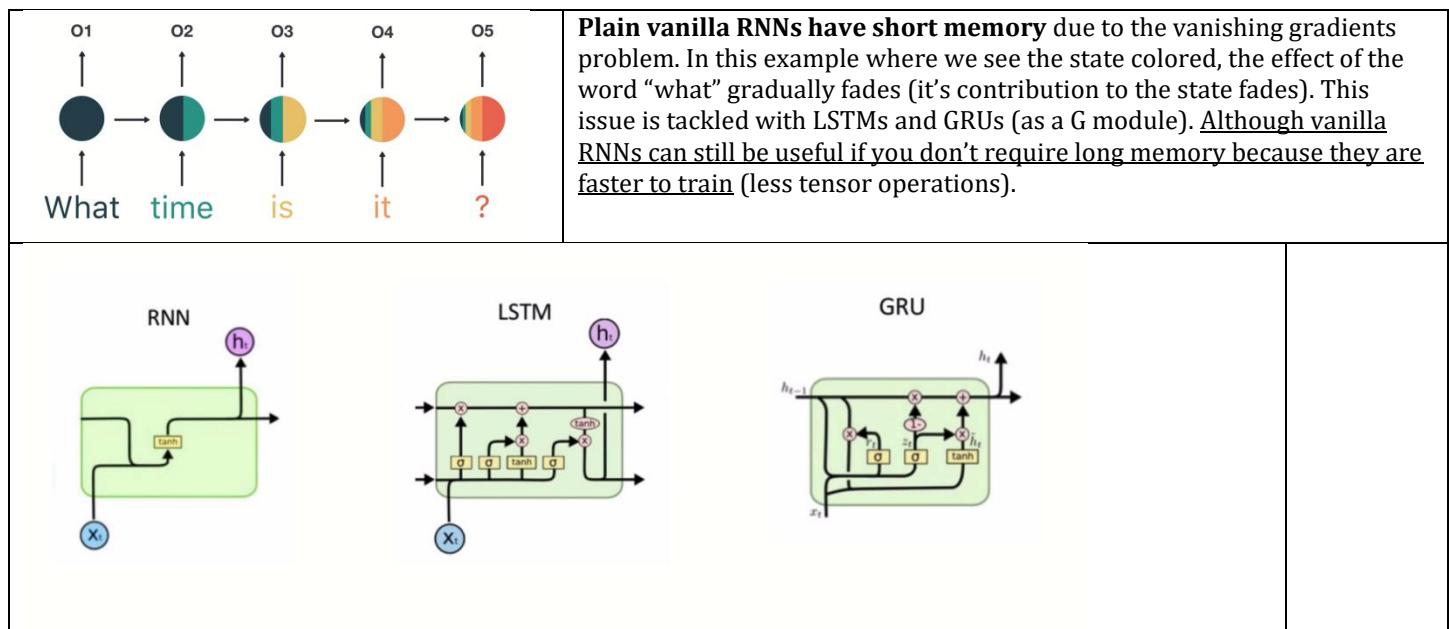
Notice that the parameters of g are the same across timesteps. So, we can think of it as a module that it is copied across the input. This means that **during training we can sum the gradients of each timestep to calculate the total gradient** w.r.t the parameters w of the g function. So, you get the first input of the sequence $x(0)$, lets say a word, and the RNN outputs a $y(0)$. You get a cost gradient that is backpropagated through G to get the gradient with respect to w . You accumulate that gradient. Then the next input comes. You have a new output and a new cost and a new gradient. You backpropagate it and add it to the previous. And so on, until the whole sequence is processed. The gradient with respect to w is the total sum of gradients. This is one epoch. Then you continue to the next epoch with a new sequence and so on.

You must think of this unfolded across time acyclic neural network of the picture as the real network you train. This unfolded network is the RNN. The larger the sequence, the larger the number of copies of the basic module, and the larger the total RNN is.

The hidden state z is a representation of the previous inputs. $z(1)$ is a representation of inputs $x(0)$ and $x(1)$ and it will affect both the next hidden state and the output of the rnn (the y).

Another way to think of an RNN is as a multilayer NN where the state z is the output of each layer. Every layer is the same. And a new input is added to each layer's output.

Traditionally it was very difficult to train large vanilla RNNs (where the G module is a sigmoid and a linear module), GRUs and LSTMs solve short memory caused by vanishing gradients, because of exploding or vanishing gradient/state. In typical ANNs the problem is significant when you have many layers. In RNNs the problem appears when you have many sequence steps (when you want to train it with large sequences). A state $z(1)$ is the output of the module g. Forget for a moment the input x . If g is a linear module that transforms z by scaling it, then the state z will explode as you go. The same with the gradient (the transpose of a diagonal matrix is the same diagonal matrix). So, the state z and the gradient of z are the components facing the explosion/vanishing problem. Me: When you have this problem, then essentially the network is trained only by some part of the input space (of the input sequence). For example, in the vanishing gradient case, the first parts of the sequence are ignored since the gradient there is close to 0. So there is no point on having large vanilla RNNs.



Ways to tackle exploding/vanishing state/gradient in RNNs (in general)

RNN tricks	A way to tackle exploding gradients is to use a sigmoid like nonlinearity in G. but that doesn't tackle vanishing gradients.
<ul style="list-style-type: none"> ▶ [Pascanu, Mikolov, Bengio, ICML 2013; Bengio, Boulanger & Pascanu, ICASSP 2013] ▶ Clipping gradients (avoid exploding gradients) ▶ Leaky integration (propagate long-term dependencies) ▶ Momentum (cheap 2nd order) ▶ Initialization (start in right ballpark avoids exploding/vanishing) ▶ Sparse Gradients (symmetry breaking) ▶ Gradient propagation regularizer (avoid vanishing gradient) ▶ LSTM self-loops (avoid vanishing gradient) 	

Some details on the reasons of vanishing gradients in vanilla RNNs

How stable state is memory

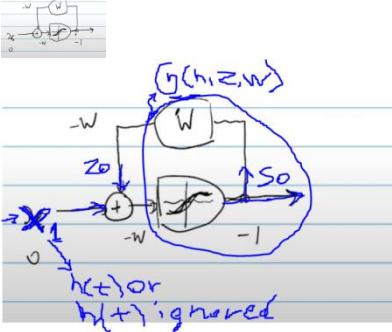
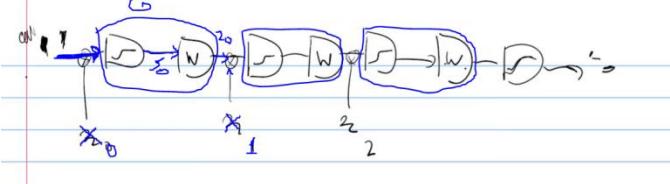
The state is the thing that remains from the previous processing and affects the current processing. Imagine that the state doesn't change after the third component of the sequence. It remains stable. This means that the next units will be affected by this state which is the result of the first 3 components of the sequence. Essentially the system stores a representation of the effect of the first 3 components and use it to affect the processing of the current component. So if a component of the sequence

needs to take into account something that happened a long time ago in the sequence, it can do it if the state could remain stable between the time of interest and now. The system can learn to do this.

Stable state means vanishing gradients

Stable state though, means vanishing gradients. Because if the first state is perturbed a bit the output state doesn't change. Which means the gradient is 0. This was published in a paper by Bengio in mid 90s and was thought to be the main reason you can't use RNNs. Because if you had memory (stable state) you wouldn't be able to train the RNN. But ultimately this conclusion was wrong. It was shown that you can have memory without a stable state. The concept is that instead of the state being stable, it is transformed linearly for example rotating it (the state vector). If you know these rotations, you can rotate it back with the opposite angles and get the state of a few steps back. Is this what GRU and LSTM G modules do?

How you keep the state stable

	<p>A memory module This is how a memory module is implemented in some electronic systems. It is implemented similarly in RNNs. Its goal is to keep the state stable.</p> <p>Assume we start with $x(0)=0$ and $s_0=-1$. The w is a scalar parameter. It is multiplied by s_0 to give $z_0=-w$. $-w$ passes through the sigmoid and gives output -1. If you are in the flat part of the sigmoid then no matter the perturbations of x, the s_0 remains stable at -1. If it receives a large positive x and will go to $+1$ and will stay there if the perturbations of x keep the sum in the flat region of the sigmoid. If the s_0 is between 0 and $+1$ it will move quickly to $+1$. If it is between -1 and 0 it will move quickly to -1. So -1 and $+1$ are the state attractors. It is a bistable system where it can settle in one of two states.</p>
	<p>You can think of a simple RNN as a sequence of modules like this where the G module is a sigmoid and a linear module. In this system, as we said, the state is bistable, and you have the vanishing gradient issue so you can't train them.</p>

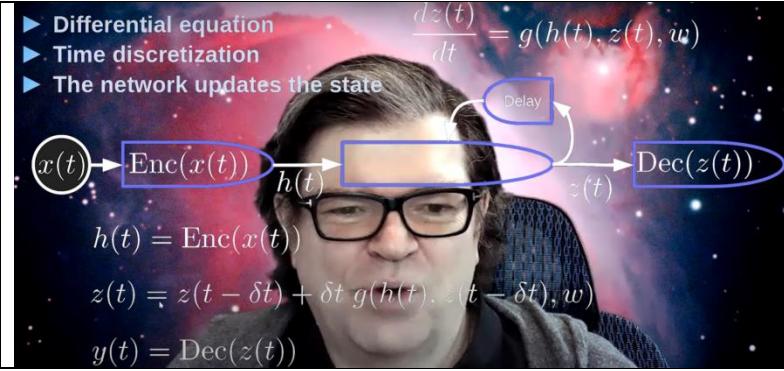
Some types of $G(x, z, w)$ modules

1. GRU
2. LSTM

But how these modules tackle the vanishing/exploding state issue? They don't have the previously described mechanism of a vanilla RNN for creating a stable state. So, the state is not stable when you use these modules. This solves the vanishing gradients problem. But how they have long memory? You can somehow undo the transformations from let's say 25 steps back, get the state at that point and use it? Maybe their internal structure allows them to learn to do this.

The main point in these modules is that they remember by default (by default the state remains intact) and they have NNs that learn to slightly modify the state when needed. These NNs act as gates. (So they are actually a version of the neural ODE). They learn what information of the state to retain and what information from the observation to use to update the state. They do this with the use of sigmoids where output close to 0 means forget and close to 1 keep. This way they can learn to remember important observations that happened long ago in the sequence. <https://www.youtube.com/watch?v=8HyCNIVRbSU>

Neural ODE (just have in mind)

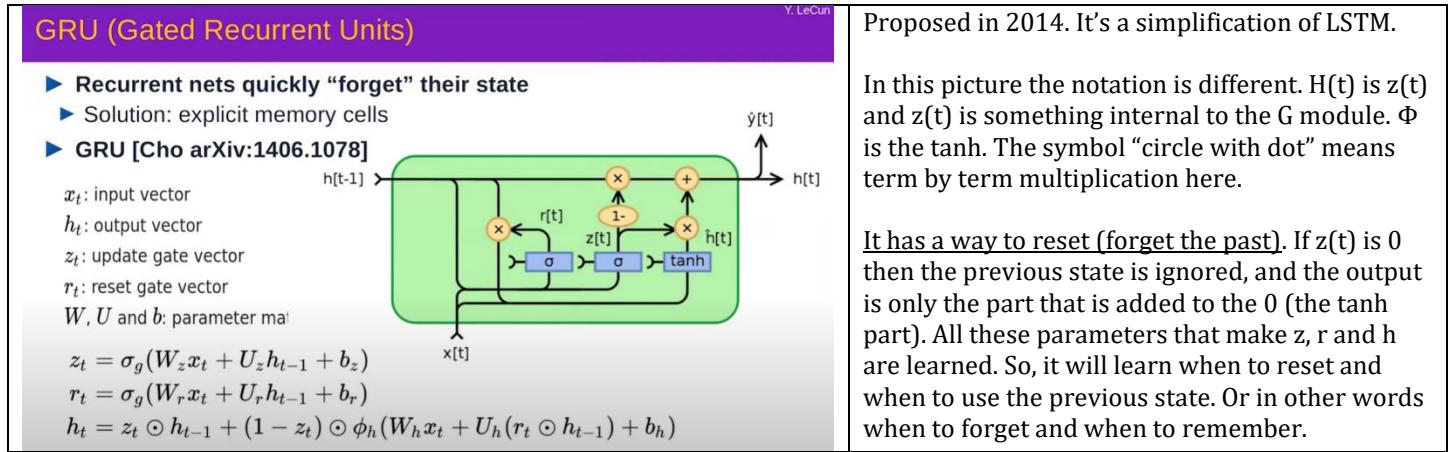


Neural ordinary differential equation. Another version of a RNN. It is used when you want to predict phenomena that obey differential equations (physics, video games, timeseries prediction etc.). For example the orbit of a throwing ball. The input data are the past position and momentum, and you want to predict the new position and momentum.

So, in these cases, the z function (the state of the RNN) is an equation that describes this natural phenomenon, the equation for position and speed and the g function is its time derivative (it shows how z changes with time). So the difference with the standard RNN is that now the function g is the derivative of z with respect to time instead of being simply z . so the module that gets h and produces z , instead of computing a new $z(t)$ in every step as before, it updates the old z ($z(t-1)$) with the g function.

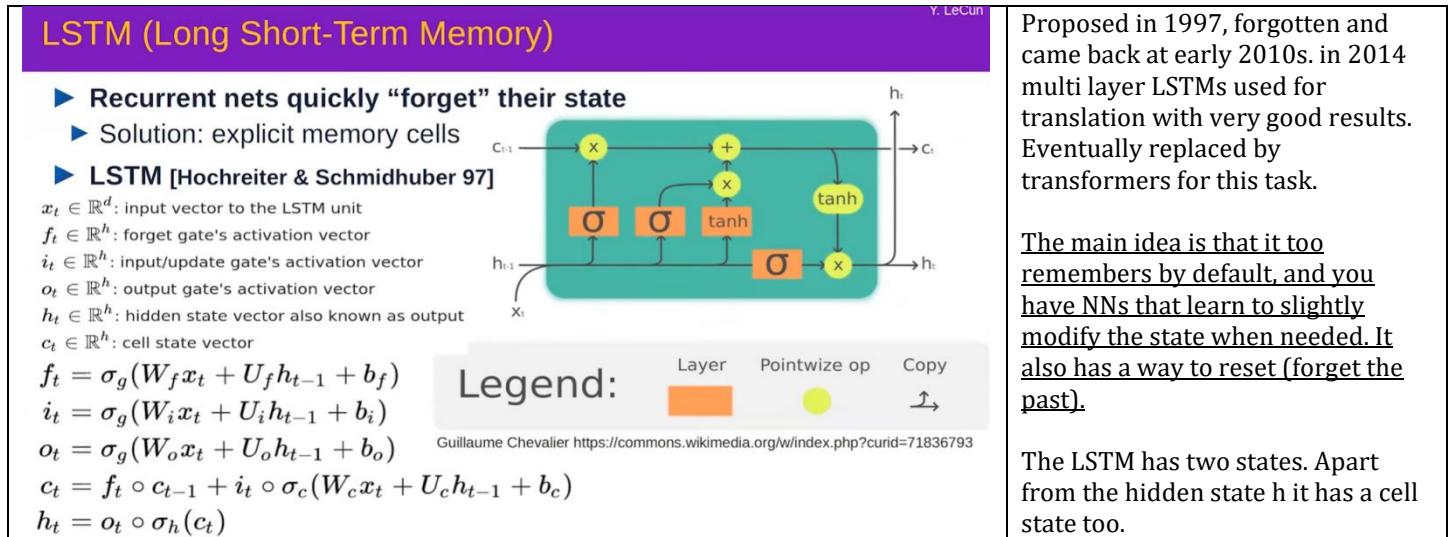
x is the observation. Z is the state of the system you are observing (position and momentum). Y is a prediction. So, you want your system to learn the differential equation that describes this phenomenon by observing a sequence of x (or x and y)

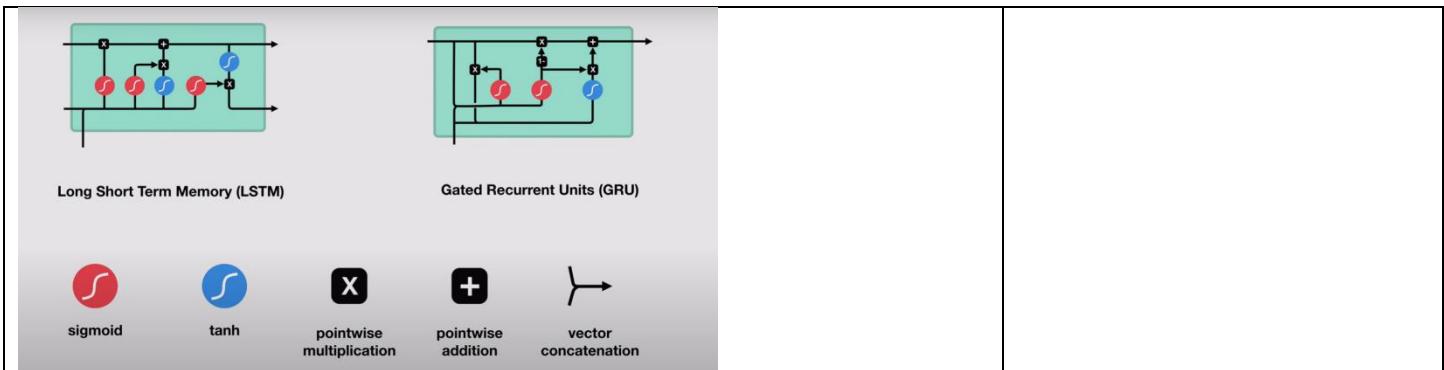
GRU



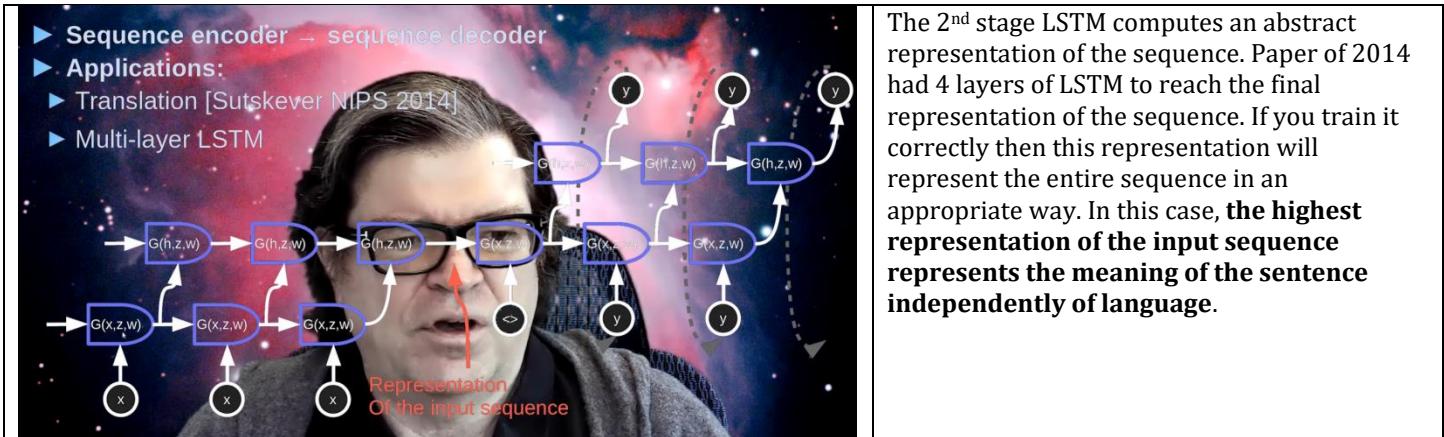
This is an elementary module that exists in PyTorch.

LSTM



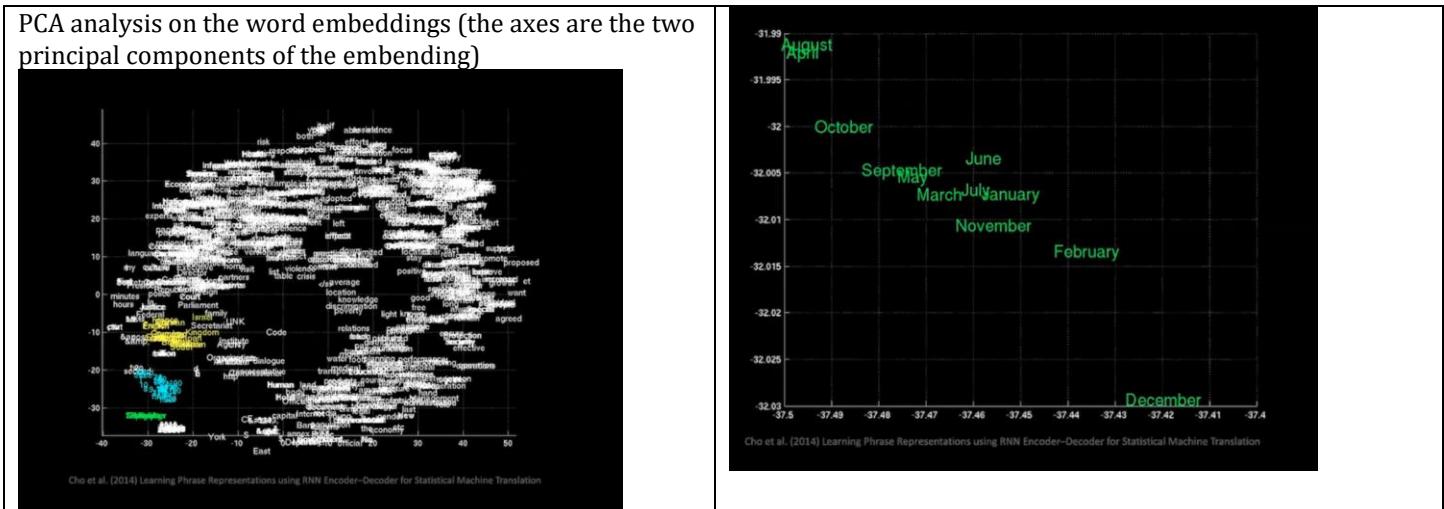


Sequential autoregressive generation of text using multi layer LSTM.



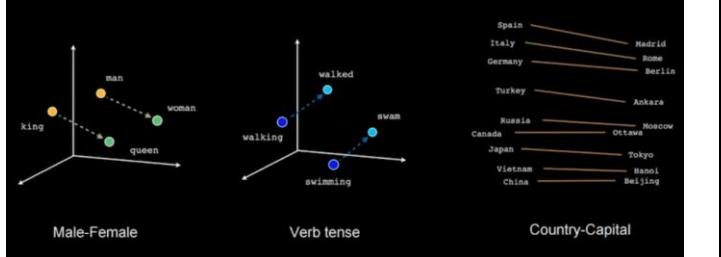
You feed this to another multi layer LSTM. It takes a null marker as the first observation, it goes through multiple layers and at the end it produces an output y which is a word in another language (a probability distribution over the words of a language, but you pick one word). Then this produced word, becomes the observation for the next time step. It is fed as input to “decoder” multi layer LSTM and so on. This system had slightly better performance than the state of the art classical translation systems based on statistical models. But it was a huge network that required a lot of gpus. So, the next evolution, which are smaller systems easier to train, was transformers which are systems that use attention.

Examining the word embeddings



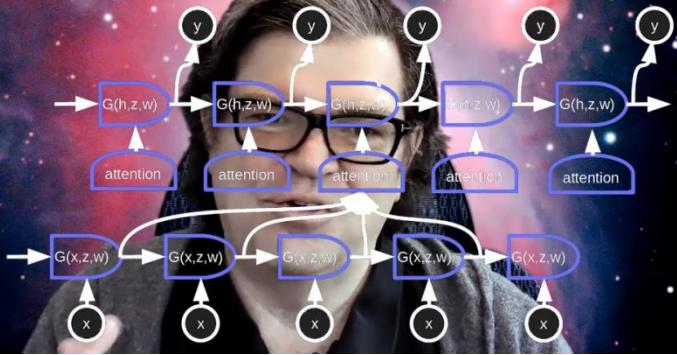
King-man+woman=queen

Distance between king and queen is same with that between man and woman.



Transformers

- ▶ Sequence encoder → sequence decoder with attention
- ▶ Applications:
- ▶ Translation [Bahdanau, Cho, Bengio ArXiv:1409.0473]

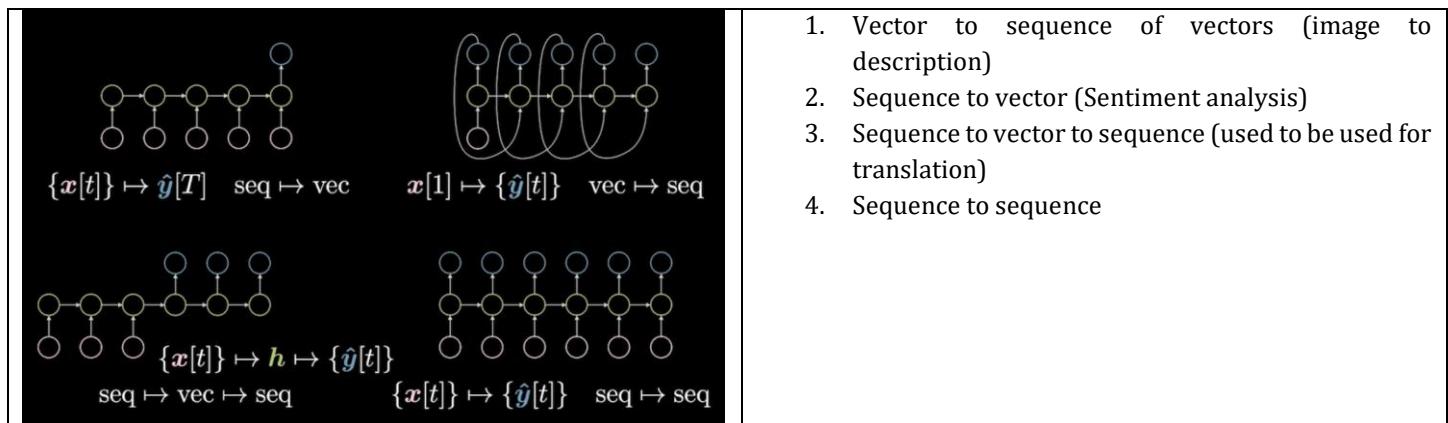


Instead of producing a single representation for the whole sequence as multi layer lstms do, the decoder part of the system, pays attention only to some part of the input sequence in order to produce its output y . this is very useful since it allows you to avoid the multi layer encoding part of the system that produces this compact representation. This attention mechanism might be very useful in translation because in some languages the word order is different. So an output word might need to pay attention to a different part of the input sequence. In a nutshell the attention module is a softmax which decides to which parts of the input sequence to switch to (mixture of experts approach).

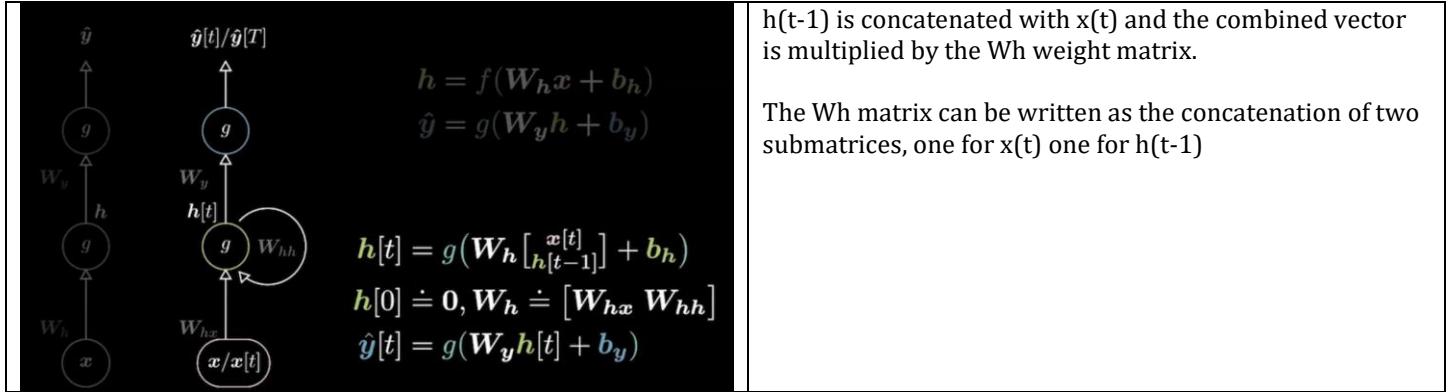
Alfredo's lab

You can apply RNNs to multi dimensional input (x sequence, y sequence, z sequence)

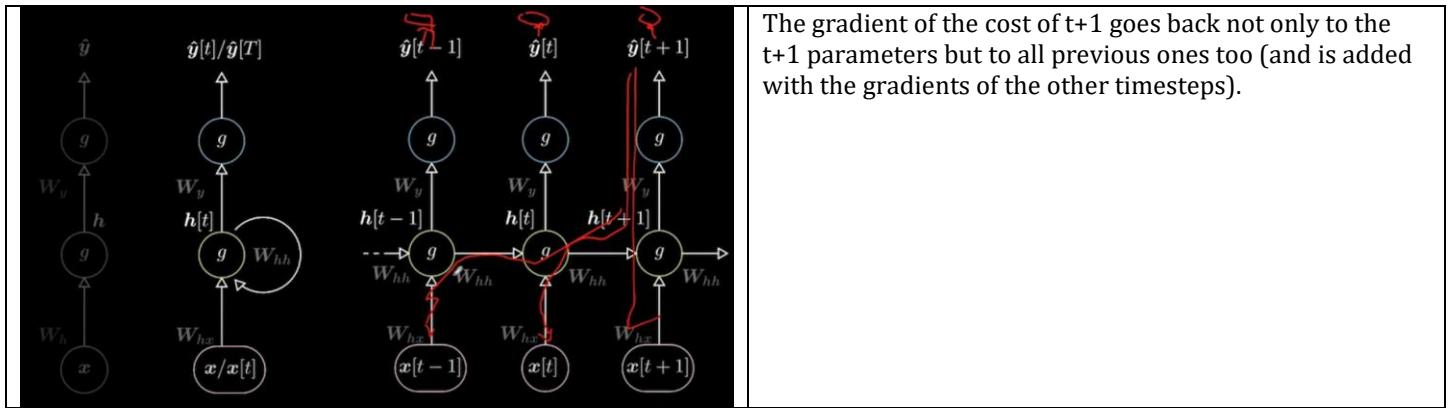
RNN applications



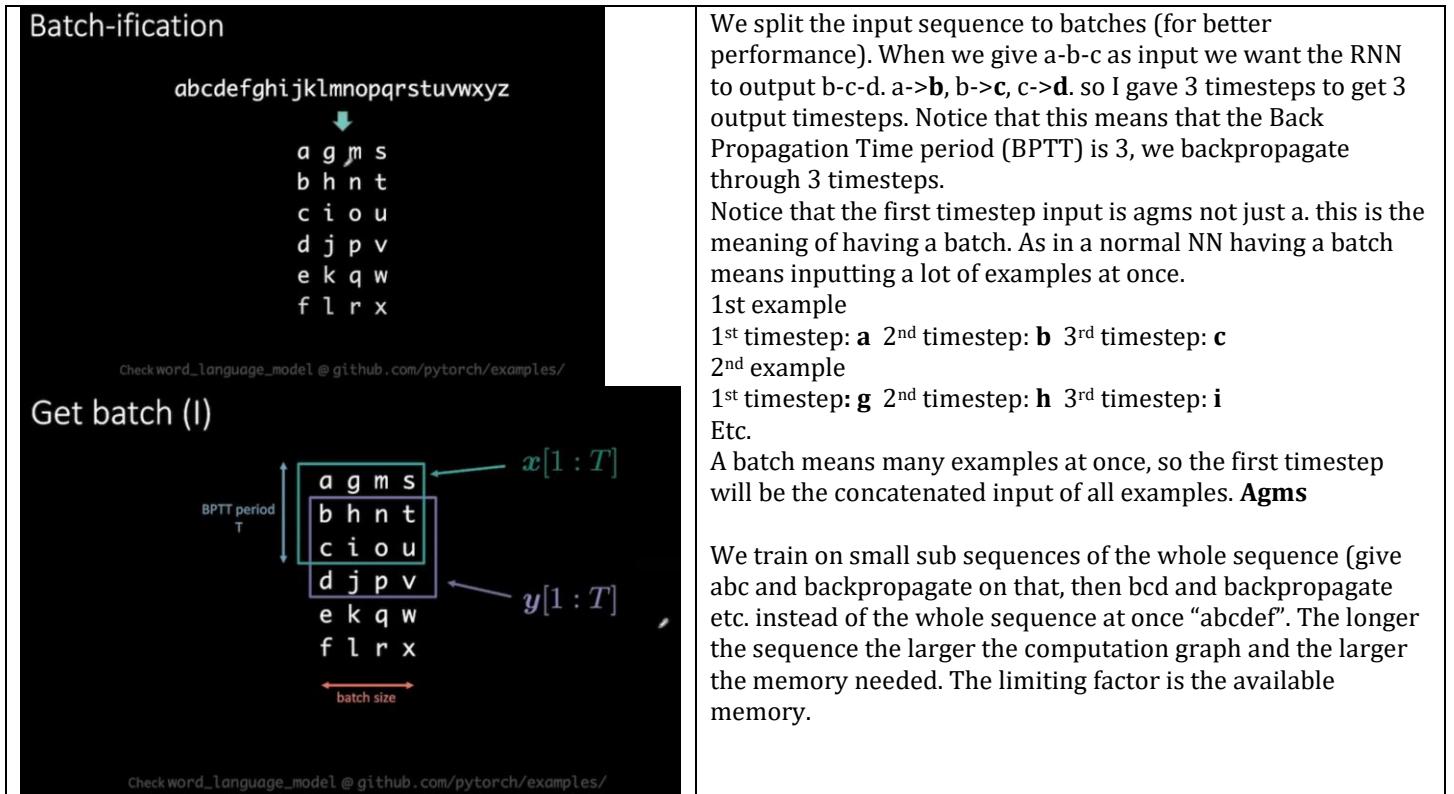
Inference in RNN

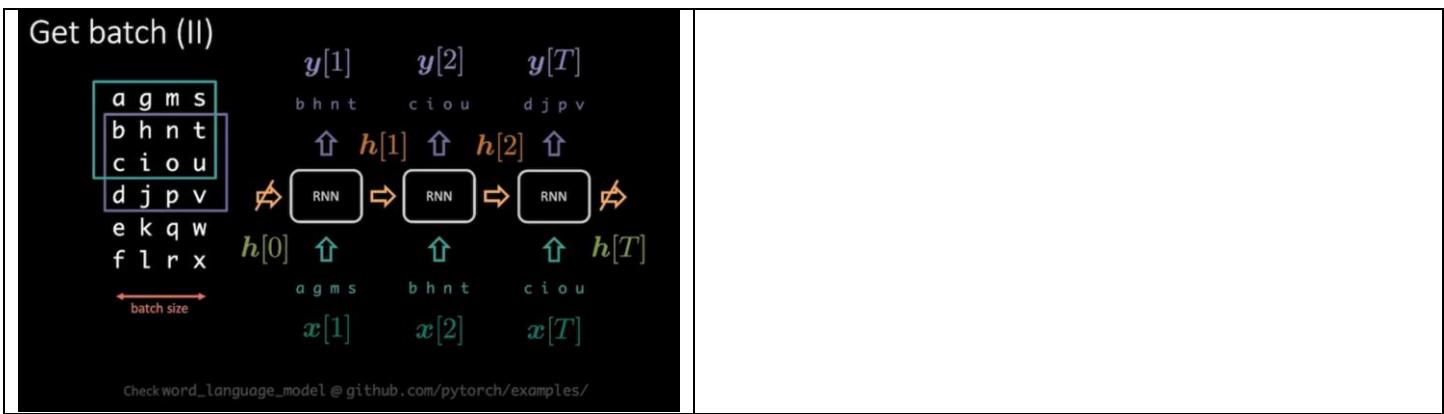


Backpropagation through time



In practice





Misc

The Kelley-Bryson method

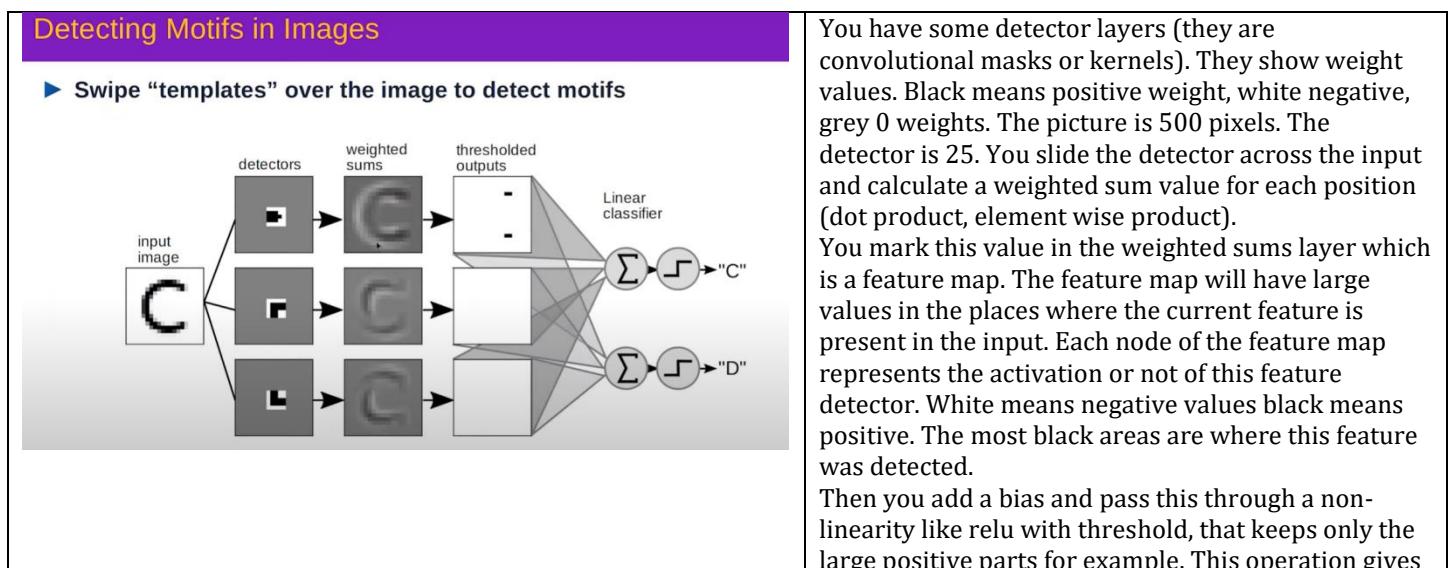
The Kelley-Bryson method (aka the adjoint state method) from 1962 is what we call backprop through time (+ gradient descent). But realizing that you could use this to train multilayer non-linear neural net, and actually making it work, didn't happen until the mid 1980s.

CNNs

Intro

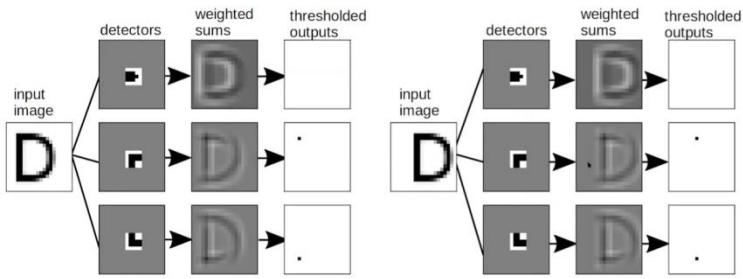
Data must have local correlation. Where two nearby values of the input vector are usually very similar with each other. For example in images there is a very high chance that if you get two nearby pixels their color will be very close. So the patterns you observe in a small area of the picture (of the size of the kernel) do not cover all the possible patterns of the image (combinations of pixels) which would be the case if the pixels were random. If you randomly permute the pixels CNN will not work properly. A fully connected net that doesn't care about topology of the input data would work better but it comes with the cost of having more parameters.

They can detect motif regardless of their location in the input.



Detecting Motifs in Images

► Shift invariance

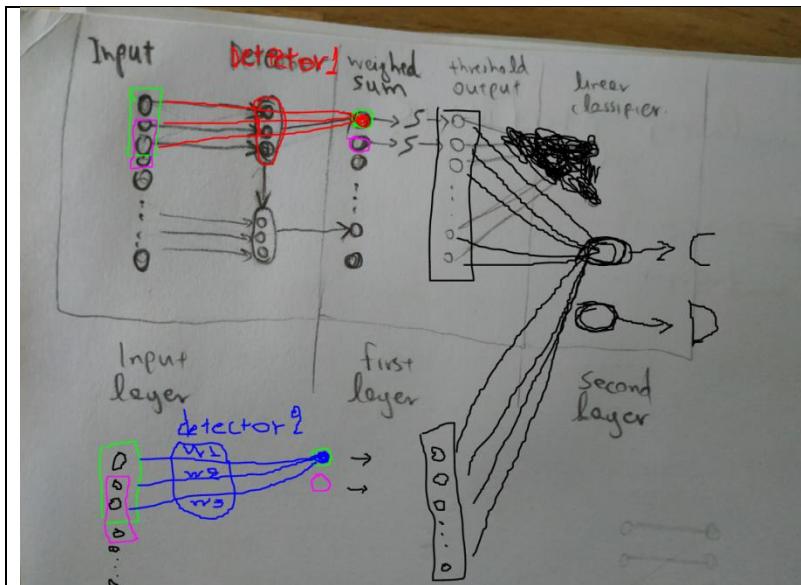


The detectors are a set of coefficients. They are called filters or convolutional kernels or masks. You have one convolution for each kernel.

a new image. You get one such image for each detector. Then you pass these resulting images of all detectors through a linear classifier. This means that in this case, the input to the linear classifier is 3 times the size of the input image (the size of the input multiplied by the number of detectors). If it has two positive areas for the first detector and no positive for the last two detectors it might learn to classify it as a C. It might compute the sum of those maximum (thresholded) values so it doesn't care where the positives are.

The detector has equal number of positive and negative weights so that when all inputs are the same it sums to 0 (the dot product with the input is 0). You can train this system with backprop and learn the feature detectors.

This process of taking a small pattern of coefficients and sliding it over the input is called discrete convolution. It is actually a different type of a linear function, different from the typical weighted sum. It is different in two ways.



This first layer is a convolutional layer. It is a function in pytorch that you can easily add to your model.

1. Weights are localized.

The nodes of the first layer are not fully connected to the nodes of the input. You can think of it like this. Each node of the first layer, only "looks" at a specific location of the input and it looks at it through a specific mask (the detector). Think of a mask as a set of weights (not nodes). Each mask will affect it differently for the same input. And for each node, instead of getting one weighted sum as in the typical linear module where you sum the contribution of all input nodes and all weights, you get one weighted sum for each mask where you only sum the input nodes of the location the layer node looks at.

2. Weights are shared

All nodes of the first layer look at their own location through the same mask. So they all use the weights of this mask. They share the same weights for each mask.

Convolution

- Definition
- convolution

$$y_i = \sum_j w_j x_{i-j}$$

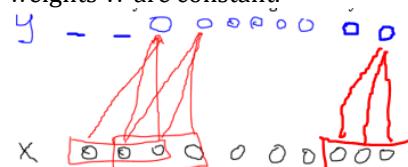
- In practice
- Cross-correlation

$$y_i = \sum_j w_j x_{i+j}$$

- In 2D

$$y_{ij} = \sum_{kl} w_{kl} x_{i+k, j+l}$$

Pytorch uses the cross-correlation formula but it calls it convolution. They are the same thing essentially. The weights W are constant.



The output is the same size of the input, minus the border effects.

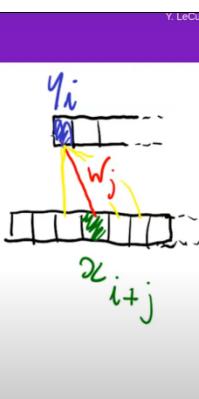
Backpropagation through convolution

Backpropagating through convolutions

- ▶ Convolution $y_i = \sum_j w_j x_{i+j}$
- ▶ (really: cross-correlation) \sum_j

- ▶ Backprop to input
 - ▶ Sometimes called "back-convolution"
- $$\frac{\partial C}{\partial x_j} = \sum_k w_k \frac{\partial C}{\partial y_{j-k}}$$

- ▶ Backprop to weights
- $$\frac{\partial C}{\partial w_j} = \sum_i \frac{\partial C}{\partial y_i} x_{i+j}$$



Convolution is a linear operation so the gradient with respect to the input vector x is a multiplication of the incoming gradient with the transpose of the weight matrix. One x value influences let's say 3 y values. You know the gradient of the cost with respect to every y . the gradient with respect to one x value (x_i) is the sum of the gradients of the y values it influences multiplied by their weights.

Gradient with respect to the weights. the gradient of one weight w_j , typically is the gradient of the y value multiplied by the x value. But in convolution the same weight is used multiple times since the weights are shared (you slide the kernel so you slide the weights) so we add the gradients of all y values the weight is connected to.

Types of convolutions

Stride and Skip: subsampling and convolution "à trous"

- ▶ Regular convolution
 - ▶ "dense"
-
- ▶ Stride
 - ▶ subsampling convolution
 - ▶ Reduces spatial resolution
-
- ▶ Skip, convolution "à trous"
 - ▶ pronounced "ah troo" (means "with holes")
 - ▶ Dimensionality reduction without loss of resolution
-

Stride

Stride of 2 means that we move the kernel by 2 steps instead of 1. It is subsampling actually. It reduces the size of the output.

A trous (dilating convolution)

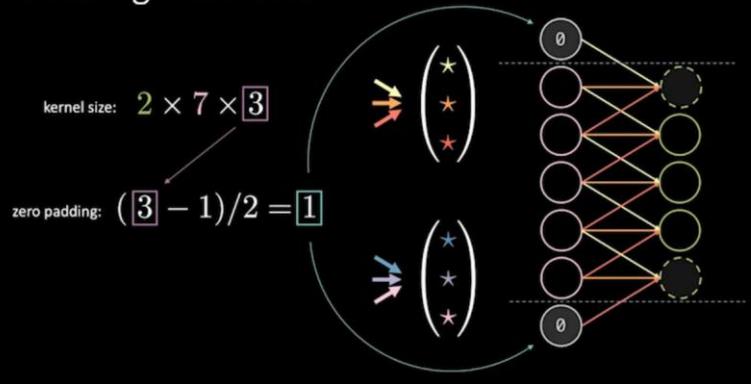
Some weights of the kernel are 0. It is useful if you want your output nodes to have a wider perception field but without increasing the number of weights (parameters) too much.

Same convolution

A same convolution is a type of convolution where the output matrix is of the same dimension as the input matrix. This is achieved with padding (see alfredo's lab)

Same convolution: A same convolution is a type of convolution where the output matrix is of the same dimension as the input matrix. It achieves this with the use of padding. (Handling border effects)

Padding – 1D data



Padding

In order not to mess the matrices size, you need to add some input units with 0 value, and perform a convolution there too. this way the size of the output is the same as the input. The 0 input unit will just contribute 0 to the result of the new output unit.

This way we pad 2 additional output units.

Notice that we prefer odd number for the kernel's receptive field so that we pad an integer number of units. This means that the number of input units will be the same as the number of output units. You have one to one correspondence. Otherwise if you have one output unit corresponding to more than one input unit you will have a blur effect.

Pooling

It eliminates a bit of information about the precise location of where a feature is detected. For example. You might have a feature detected in the first node and in the second node. The pooling will be activated no matter if the feature is in the first or second node. So it eliminates some information about the precise location of the feature. This adds to the robustness of convnets with respect to the location of features.

Pool: aggregation values
permutation invariant

$$f(x_1, x_2, \dots, x_n) = f(x_{p(1)}, x_{p(2)}, \dots, x_{p(n)})$$

$$y = \frac{1}{k} \sum_{k=1}^K x_k \quad \text{average pooling}$$

$$y = \left(\sum_n x_n \right)^{\frac{1}{p}} \quad L_p \text{ pooling}$$

$$y = \max_n (x_n) \quad \text{max pooling}$$

$$y = \frac{1}{p} \log \sum_n e^{B_n x_n} \quad \text{logsumexp pooling}$$

You aggregate neighboring values by subsampling. You can pool over 4 units and shift over 2 so there might be overlap in the pooling.

Subsampling ratio = 2 means that you go from n values to n/2 values if you have no overlap in the pooling.

The pooling operation is an aggregation which must be permutation invariant. This means that it should not be influenced by change in the order of the input values.

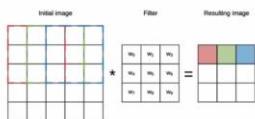
The most famous pooling function is Max pooling.

Logsumexp pooling. B → 0 then it is average pooling. B → inf is max pooling. B → -inf is min pooling.

Kernels must be of odd number (for example 5*5 or 7*7 in an image) so that you can have a central point (a central pixel)

Calculate the size of the output image after a convolution

Convolutional Layer



$$(W-F+2P)/S + 1$$

example: 5x5 input, 3x3 filter, padding=0, stride=1

$$\begin{aligned} (5-3+0)/1+1 &= \\ 2/1+1 &= 3 \\ \rightarrow & 3 \times 3 \end{aligned}$$

This is the formula

$$n_{out} = \left\lceil \frac{n_{in} + 2p - k}{s} \right\rceil + 1$$

n_{in} : number of input features

n_{out} : number of output features

k : convolution kernel size

p : convolution padding size

s : convolution stride size

Convolution Arithmetic

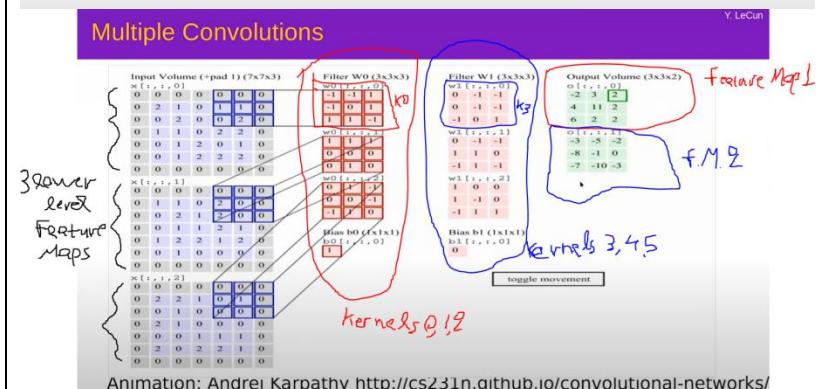
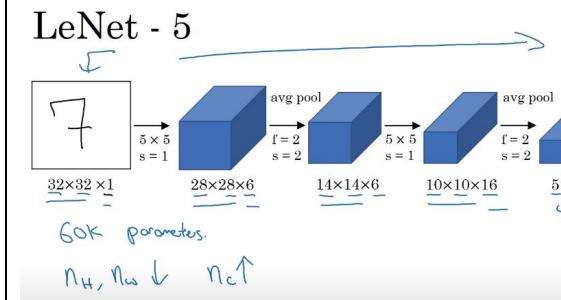
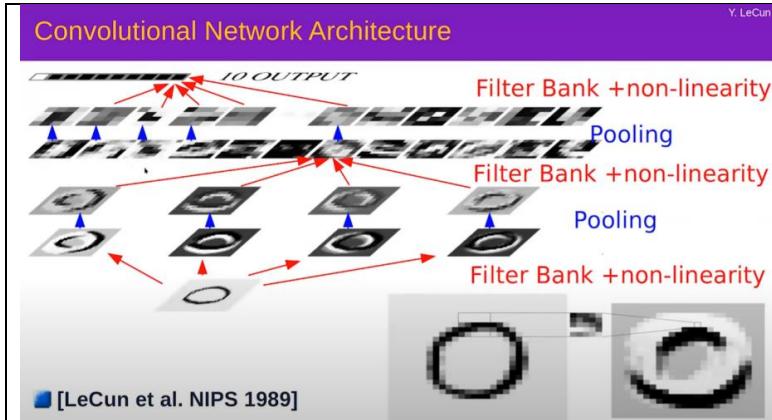
1d, 2d and 3d convolutions

In 1D CNN, kernel moves in 1 direction. Input and output data of 1D CNN is 2 dimensional. Mostly used on Time-Series data.

In 2D CNN, kernel moves in 2 directions. Input and output data of 2D CNN is 3 dimensional. Mostly used on Image data.

In 3D CNN, kernel moves in 3 directions. Input and output data of 3D CNN is 4 dimensional. Mostly used on 3D Image data (MRI, CT Scans, Video).

The architecture of a CNN



In the animation, the higher level feature map 1 is the sum of the individual feature maps produced by the dot product of the input with each one of the kernels 0, 1 and 3. These "intermediate feature map components" are not shown

Each feature map after pooling, passes through a different kernel. The result for each kernel is added with the others. This sum is the feature map of the next layer. This summation allows the second layer feature maps to detect combinations of features detected in the previous layer.

You can think of the feature maps of a layer as channels (see alfredo's lab). Essentially, the 2d input signal (black and white input image) is transformed to a 3d signal where the third direction is the number of feature maps (so the $32 \times 32 \times 1$ is transformed to a $28 \times 28 \times 6$ in the LeNet example). It's getting smaller because at the time there was no use of padding in convolution to tackle the border effects). The kernels of the next layer are applied to a multi-channel signal. This means (as described in alfredo's lab) that each kernel sees at each layer with a different set of weights (as if it is a different kernel for each channel) and you get one feature map for each channel. Then you add these channel feature maps to get a new combined feature map on the next layer. You repeat for every kernel. This example has 12 kernels in the second layer (so each kernel sees at 4 channels and that gives you a total of $12 \times 4 = 48$ sets of weights as I understood).

Suppose that you use two kernels in the first layer. One detects vertical edge and the other horizontal edge. You get two feature maps. Then each feature maps passes through another "kernel" (the set of weights of the kernel for this channel) and produces two new feature maps that are added with each other to produce a common new feature map. These new "kernels" might end up being a corner (an intersection between a horizontal and a vertical edge) in which case the common feature map represents the activation of corner features.

Usually the pooling does subsampling so the pooled feature maps are low resolution versions of the unpooled.

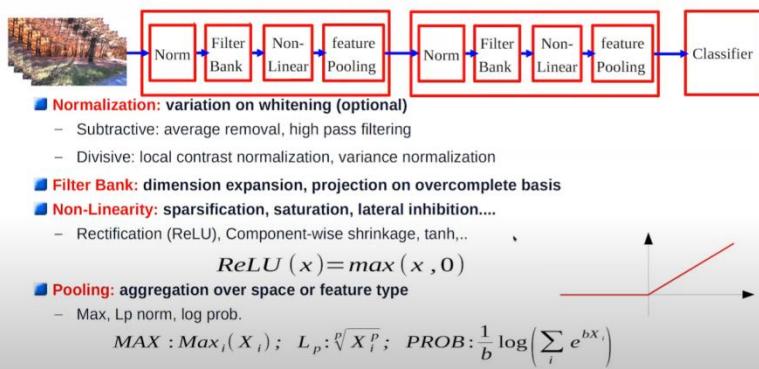
Notice that at the final layers you get representations of the input with almost no spatial information about it, the reason is that you apply subsampling in each layer and summation of feature maps. So a unit of the final feature map is influenced by a big part of the input.

Convolutional Network (vintage 1990)

Filters-tanh → pooling → filters-tanh → pooling → filters-tanh



Overall Architecture: multiple stages of
Normalization → Filter Bank → Non-Linearity → Pooling



In the final layer the convolution kernel has the same size as the height of the last feature map so the resulting feature map is 1d and not 2d.
 This collection of the final (1d in this case) feature maps is the representation of the input image and it is the input to the classifier (which might have multiple layers)

```
def __init__(self):
    super(LeNet5, self).__init__()

    self.convnet = nn.Sequential(OrderedDict([
        ('c1', nn.Conv2d(1, 6, kernel_size=(5, 5))),
        ('relu1', nn.ReLU()),
        ('s2', nn.MaxPool2d(kernel_size=(2, 2), stride=2)),
        ('c3', nn.Conv2d(6, 16, kernel_size=(5, 5))),
        ('relu3', nn.ReLU()),
        ('s4', nn.MaxPool2d(kernel_size=(2, 2), stride=2)),
        ('c5', nn.Conv2d(16, 120, kernel_size=(5, 5))),
        ('relu5', nn.ReLU())
    ]))

    self.fc = nn.Sequential(OrderedDict([
        ('f6', nn.Linear(120, 84)),
        ('relu6', nn.ReLU()),
        ('f7', nn.Linear(84, 10)),
        ('sig7', nn.LogSoftmax(dim=-1))
    ]))

    def forward(self, img):
        output = self.convnet(img)
        output = output.view(img.size(0), -1)
        output = self.fc(output)
        return output
```

Visual cortex

The optical nerve has 1 million fibers. So the input to your cortex is a vector of 1million units (or an image of 1 million pixels). Actually the resolution of the retina is higher around 60 million pixels but it has a few layers of neurons before the optical nerve that subsample the image to 1 million pixels so that it can pass through the optical nerve.

There are two path ways in vision, the ventral stream that deals with object recognition (this is where convnets apply) and the dorsal stream that deals with vision for action (analysis of motion and spatial relations, calculate relative positions so that you can grab an object etc.)

It needs 100ms (1/10 sec) for a visual signal to be processed by the ventral stream (to have an output). It needs to be quick since it has to do with predators fast detection etc.

Invariant equivariant

Local translation invariance

Applications

Convents for image generation

Convent Input data

- Signals that comes to you in the form of (multidimensional) arrays.
- Signals that have strong local correlations.
- Signals where features can appear anywhere
- Signals in which objects are subject to translations and distortions.
- 1D ConvNets:** sequential signals, text
 - Text, music, audio, speech, time series.
- 2D ConvNets:** images, time-frequency representations (speech and audio)
 - Object detection, localization, recognition
- 3D ConvNets:** video, volumetric images, tomography images
 - Video recognition / understanding
 - Biomedical image analysis
 - Hyperspectral image analysis

Time-frequency representations (for example spectrogram)

The value (depicted as a color or more precisely a grayscale value) is the energy of the signal in that particular time for that particular frequency band.

Input data can be timeseries or multiple timeseries for example a multi-channel one dimensional array. Instead of one item of the timeseries coming each time you have many parallel items each one belonging to a different "channel" (a different timeseries). So it is actually a 2d array.

3d arrays. For example data from lidar. Each point has x, y and z. a colored image has 3 channels R, G, B is also a 3d array (width * height * 3)

Have in mind pytorch 3d which deals with these arrays.

Videos are 3d arrays too. and if they have color they are 4d arrays.

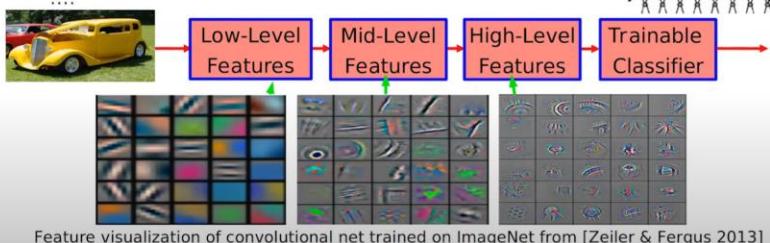
Volumetric images (voxels)

The world is compositional

Convolutional networks learn hierarchical representations

Upper-layer representation are at a coarse spatial scale

Compositional hierarchies In physics: Renormalization group theory; Multi-scale entanglement renormalization ansatz (MERA);
....



We see the input image that will maximally activate a particular neuron of a layer somewhere within the CNN. So these mid and high level features are not kernels, they are input images. The reason I guess is that the output of a unit of a higher layer is a combination of more than one kernels.

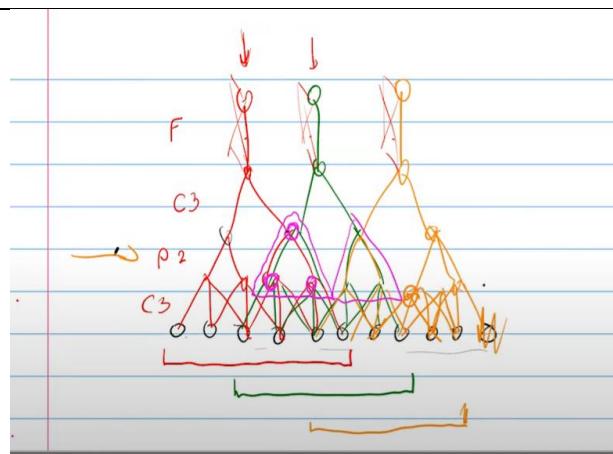
Gratings ($\sigma\chi\alpha\rho\epsilon\varsigma$)

Object detection and Multiple object detection



Object Localization algorithm locates the presence of an object in the image and represents it with a bounding box. It takes an image as input and outputs the location of the bounding box in the form of (position, height, and width).

Sliding a trained CNN



You don't actually slide the CNN, because you don't want to recalculate the "common" pink states. So you just apply the same convolution to the whole input and this means that you don't recalculate anything. Everything is calculated one time.

Both Multiple object recognition and Object detection use the same technique. **They slide a trained CNN over the input.** And this operation boils down to just applying the same convolution that the CNN does, to the whole input. so, it is as if you have one big CNN that processes the whole input although it actually isn't. in reality you have one small CNN that is sliding over the input. The size of sliding is determined by the scale of the pooling operation. If you have one pooling layer subsampling by a factor of 4 then this means that each output unit looks at a view of the input which is shifted by 4 units. But each output unit is actually

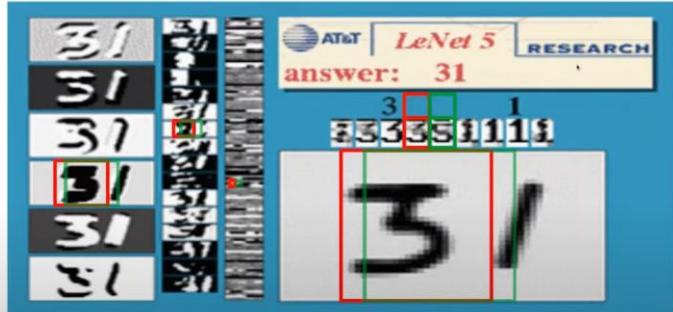
one "distinct" CNN (the one that slides over the input) and it slides over it by 4 units.

Notice that the last layers after the convolutions (C3 to F here), the layers that classify the CNN output essentially, are not fully connected but since each belongs to the same CNN, the weights are identical, and so it is as if you have a convolution with a kernel of size 1 by 1.

size 1 by 1.

Multiple object recognition

Sliding Window ConvNet + Weighted FSM



More on the structured prediction lecture

The concept is this: To detect multiple objects in an input you use one CNN trained to detect a single input and you slide it over a larger input. but you have to train it in a specific way so that it only detects the object if it is in the center of the view (if it is on the side you train it to not produce any output, if there are two objects on the sides you train it to detect nothing etc.).

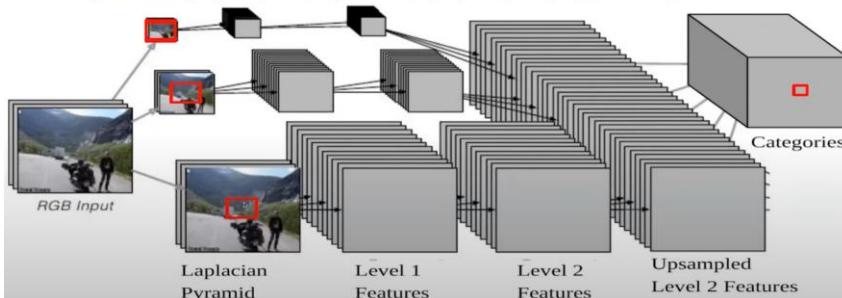
Then we slide it over a larger input that contains multiple objects. You use the same CNN without any retraining. So this process is cheap. Then each CNN produces a class it is surer about. You then apply non maximum suppression (NMS) between neighboring CNNs with the same output, to select the CNN which is more certain. Non maximum suppression is gradually replaced by trainable non maximum suppression

Object detection

Scene Parsing/Labeling: Multiscale ConvNet Architecture

Each output sees a large input context:

- ▶ 46x46 window at full rez; 92x92 at ½ rez; 184x184 at ¼ rez
- ▶ [7x7conv]->[2x2pool]->[7x7conv]->[2x2pool]->[7x7conv]->



Because you don't know If face is in a **different scale** than the CNN has trained to identify, you subsample the image and reapply the sliding CNN. Eventually there will be a scale where the face will be at the right size and the detector will fire.

Here we have a CNN applied at multiple scales of the same image. (The image is subsampled). Notice that the features of all scales are combined before going for classification.

Semantic segmentation

Semantic Segmentation with ConvNets [Farabet 2012]

Y. LeCun

- ▶ 33 categories
- ▶ 50ms per frame on a dedicated FPGA in 2011.

If you can do object detection you can do semantic segmentation as well. Essentially, you decide for each pixel if it belongs to an object or not (or at what object it belongs, on category level Semantic segmentation). You do it by sliding a CNN over the input and label the central pixel of the CNN (of the sliding window).

Some famous CNN models

Error Rate on ImageNet

Y. LeCun

▶ Depth inflation

Year	Model	Layers	Error (%)
ILSVRC'10	shallow	-	28.2
ILSVRC'11	-	-	25.8
ILSVRC'12	AlexNet	8 layers	16.4
ILSVRC'13	-	-	11.7
ILSVRC'14	VGG	19 layers	7.3
ILSVRC'14	GoogleNet	22 layers	6.7
ILSVRC'15	ResNet	-	3.6
ILSVRC'16	Ensemble	-	2.9

(Figure: Anirudh Koul)

ImageNet human performance: 5.1% top-5 classification error (expert #1), but 2.4% optimistically.

CNNs really shine when you have many categories and many training samples, for example on ImageNet

After AlexNet in 2012 which used CNN on GPUs (trained on 2 GPUs actually) all CV shifted to CNNs and there is a great improvement since.

The second graphic is a bit outdated. In 2021 the top 1 accuracy is above 90%.

Have in mind ResNet (2016, Microsoft Research) which is an important improvement. They make a group of 2 or 3 layers compute the identity function. Which means that their output is essentially identical with their input. Then they have a neural net that calculates the deviation from the identity function (the residual) as they call it. How is this useful?

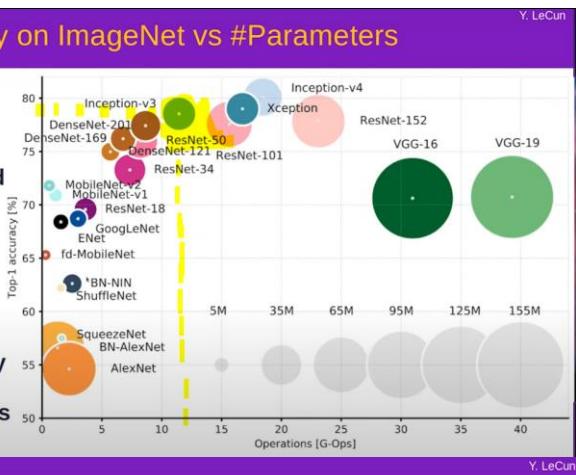
The most used architecture for object recognition is ResNet-50. It is a class in PyTorch you can just use it.

GOPS vs Accuracy on ImageNet vs #Parameters

► [Canziani 2016]

► ResNet50 and ResNet100 are used routinely in production.

► Each of the few billions photos uploaded on Facebook every day goes through a handful of ConvNets within 2 seconds.



ResNet

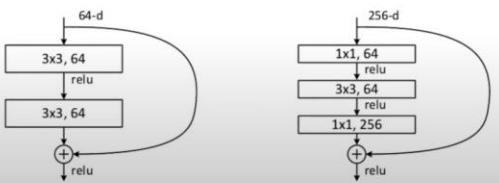
► K He, X Zhang, S Ren, J Sun: "Deep residual learning for image recognition", arXiv:1512.03385, (2015), Proceedings of CVPR 2016.

► 71,000 citations as of Feb 2021.

► The most cited paper in all of science over the last 5 years.

► Factoid: the 2nd most cited is "Deep Learning", YLC, Bengio, Hinton, Nature 2015 with 35,500 citations.

► Idea: make each layer be the identity function by default.



► Neural net computes "residual" non-linear function.

► Good with many layers

Example: ResNet-34, ResNet-50

► ResNet-50 is the workhorse of modern image recognition.

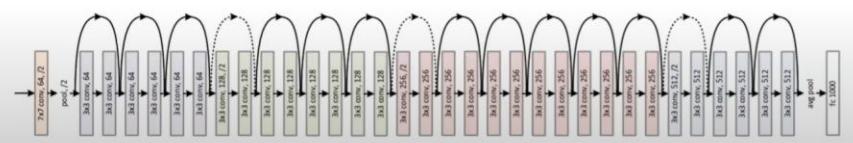
► Standard implementation in PyTorch

► Many detection systems use a so-called "standard ResNet-50 backbone"

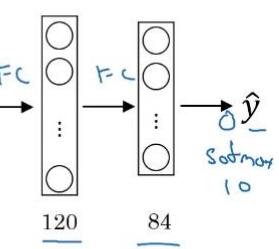
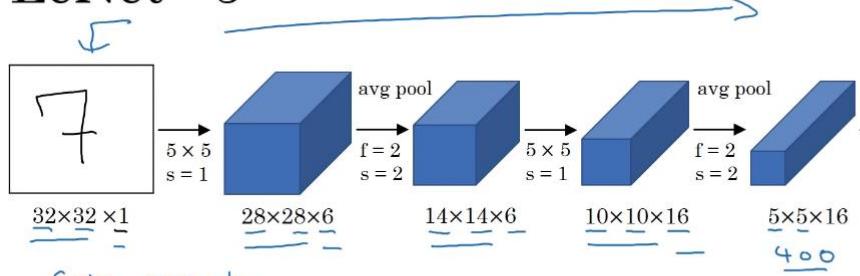
► Many variants since 2015: ResNext, DenseNet, EfficientNet.....

► Leaderboard of various architectures on ImageNet:

► <https://paperswithcode.com/sota/image-classification-on-imagenet>



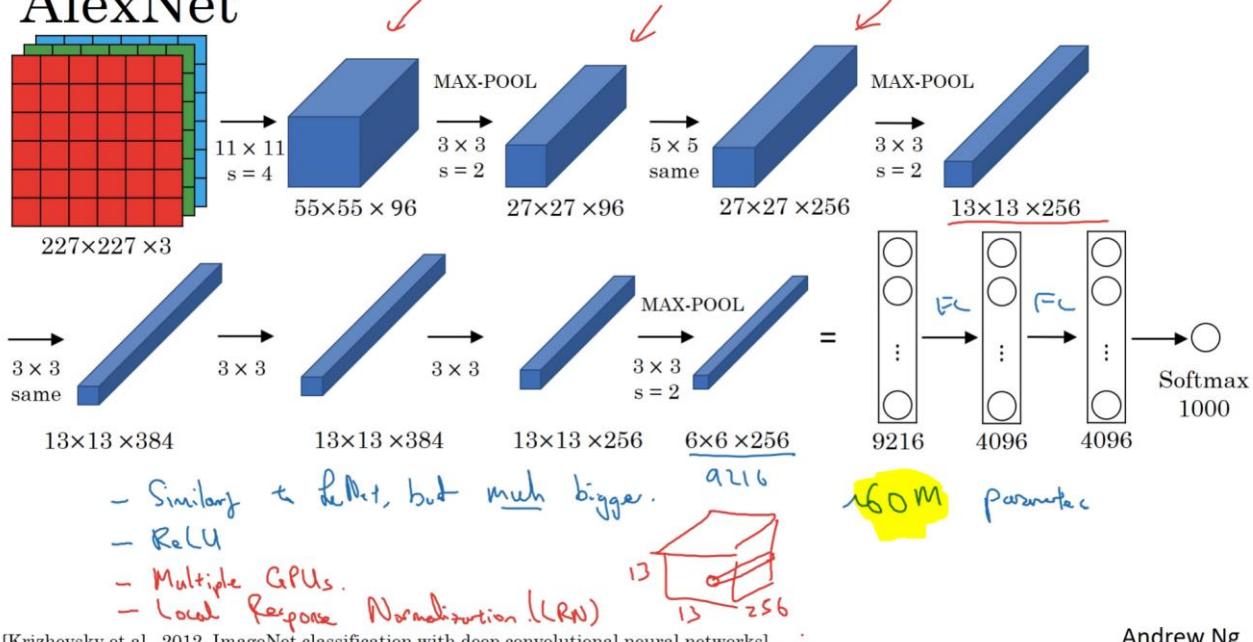
LeNet - 5



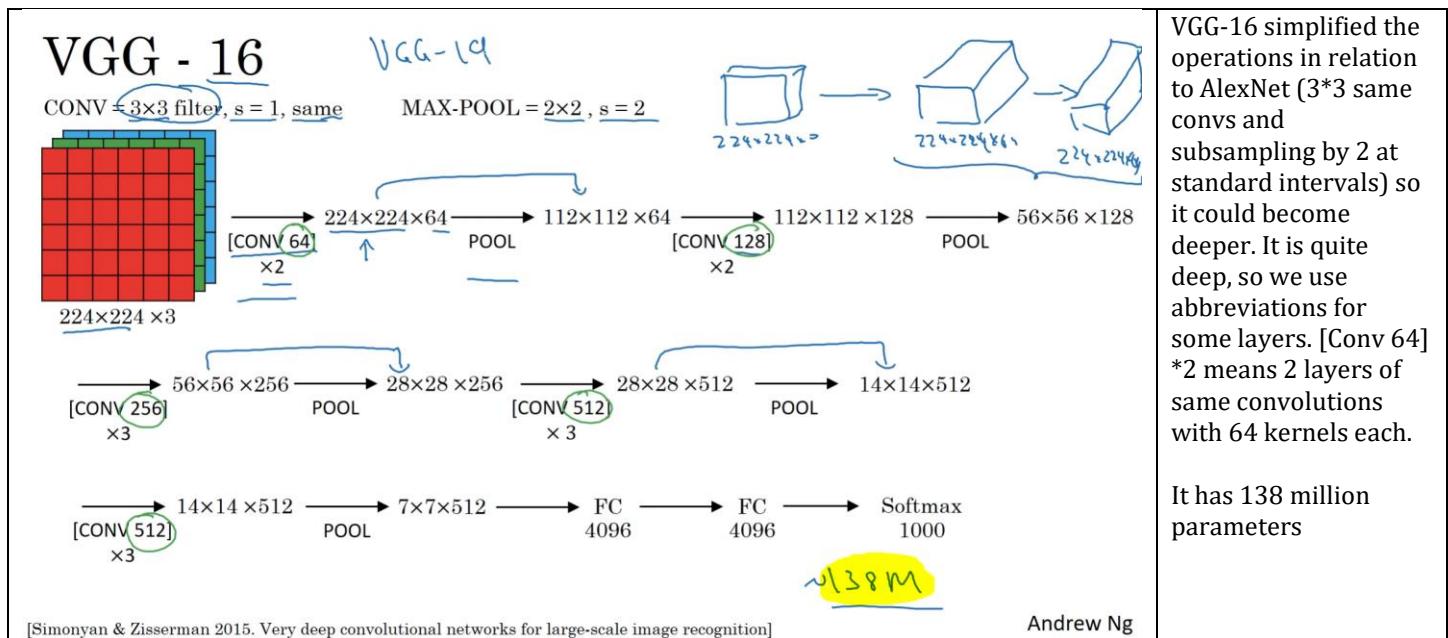
FC is an abbreviation for Fully Connected.

It used sigmoid instead of RELU back then

AlexNet

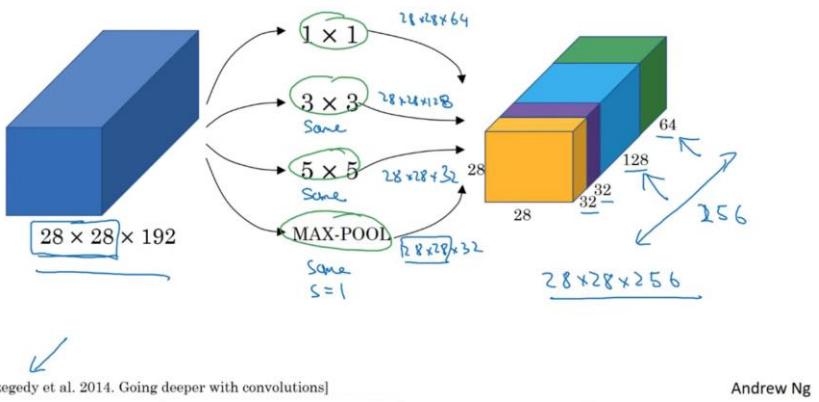


Essentially the whole concept is that as you progress in the network you reduce the resolution of the signal (subsampling) but increase the number of channels. The increase in number of channels means that you combine features to learn more complex feature as you go deeper. Down sampling means that the deeper you go, the less important is the position of the features of the input image. You might don't care where the wheels of the car are exactly in the image, just that there are some wheels (and the other features of the car in the image) to classify the image as a car.

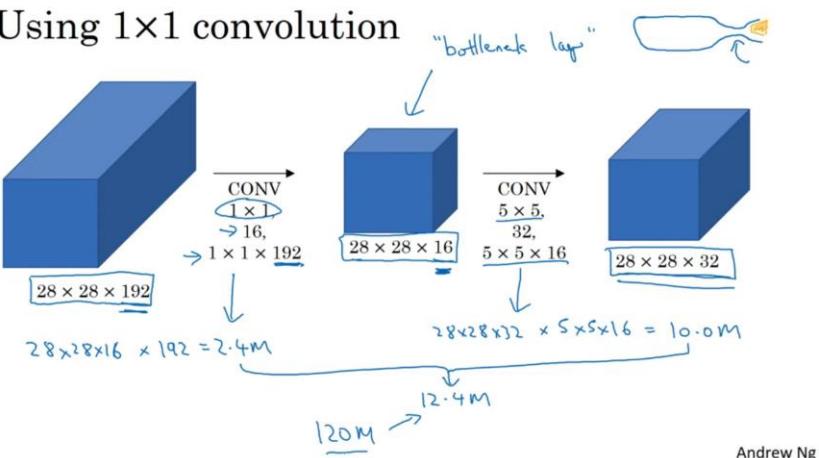


Inception (or GoogleNet)

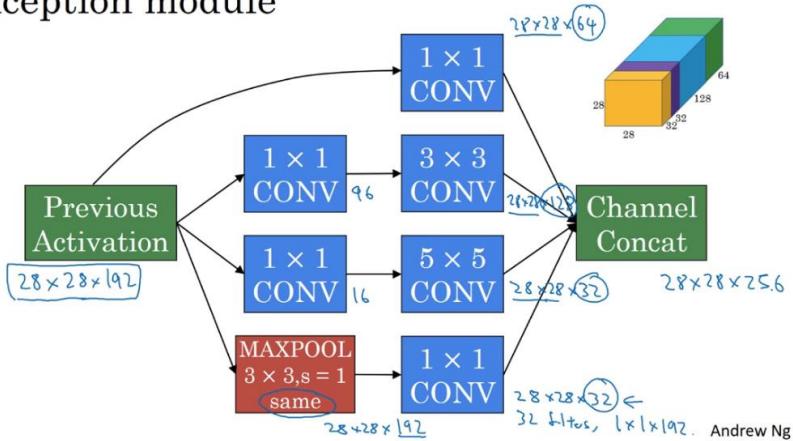
Motivation for inception network



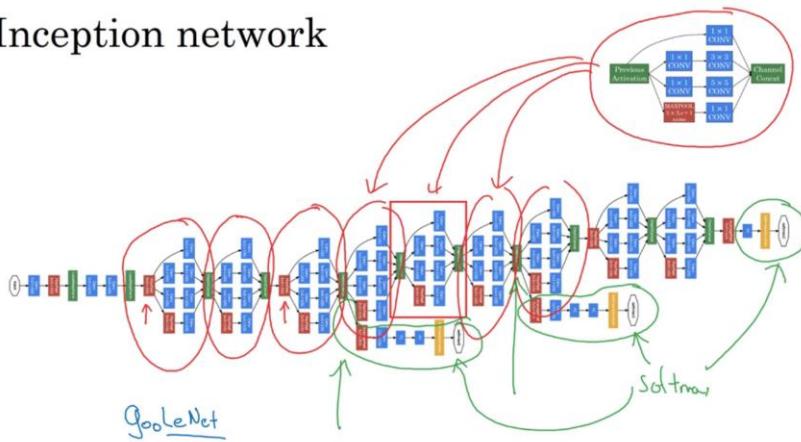
Using 1×1 convolution



Inception module



Inception network



Instead of choosing what convolution to apply in each layer (1×1 , 3×3 , etc.) you apply them all! you apply a conv 1×1 which gives you a $28 \times 28 \times 64$ output. Then you apply a 3×3 which gives you a $28 \times 28 \times 128$ output and you combine it with the previous output to form a new combined output. You can also add a pooling component to the concatenated output. Notice that it is a same type pooling so that the output is of the same dimensions in x and y.

The problem of course is that this architecture requires a lot of computation resources. But, there is a trick that can reduce the computation by a factor of 10. You use a 1×1 convolution in between.

For example instead of applying the 5×5 conv in the initial input, you first pass the input through a 1×1 conv with a smaller number of kernels so that you end up with a signal with smaller channels. Then you apply the 5×5 convolution to that smaller representation of the input. this saves you a lot of computations and its proved to be efficient.

This intermediate representation of the input is usually called the bottleneck layer.

The inception network is a sequence of inception modules.

The parts circled with green are some side branches that take some intermediate encodings and pass it through a classifier to make a prediction for the output label. So as I understand, you get more than one cost values that you back propagate at a backward pass. This is an extra push to the intermediate encodings to make correct predictions themselves instead of relying just to the final layer. This trick, seems to have a regularization effect to the network and prevent it from overfitting.

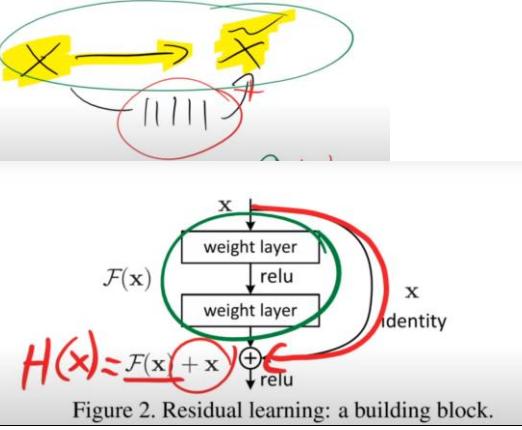
ResNets

One of the most influential papers in DL (very well written). It allowed for much deeper networks. Nowadays residual connections are everywhere, not only in CNNs but on transformers and elsewhere.

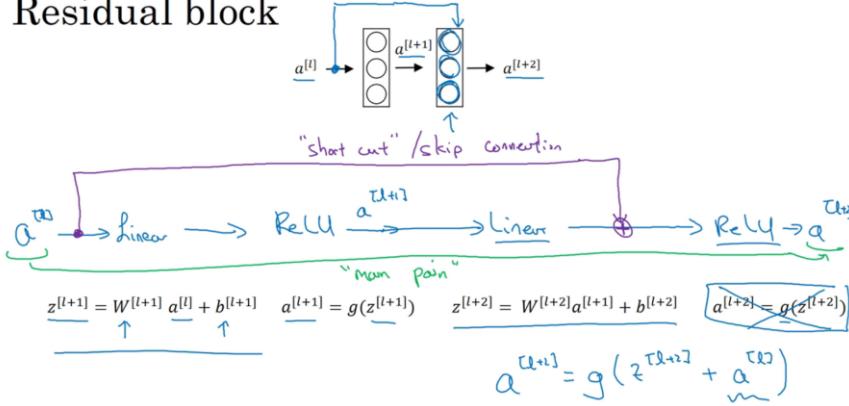
In a nutshell

There is a problem when you keep adding more layers (30+). The NN error increases in relation to shallower NN. This is a surprise. The reason is vanishing gradients. The solution reminds a bit of LSTMs. Essentially we retain the input x by default and only learn small perturbations to it.

<https://www.youtube.com/watch?v=GWt6Fu05vol> an excellent description

 <p>Figure 2. Residual learning: a building block.</p>	<p>Make x being retained by default and have the intermediate layers learn a small perturbation to x, so you get $x_{\hat{}}$. It reminds me of the LSTMs concept.</p> <p>They also have less parameters in relation to shallower models like VGG-19 because the layers in between the residual connections can be smaller.</p>
<h3>3.4. Implementation</h3> <p>1 Our implementation for ImageNet follows the practice in [21, 41]. The image is resized with its shorter side randomly sampled in [256, 480] for scale augmentation [41]. A 224×224 crop is randomly sampled from an image or its horizontal flip, with the per-pixel mean subtracted [21]. The standard color augmentation in [21] is used. We adopt batch normalization (BN) [16] right after each convolution and before activation, following [16]. We initialize the weights as in [13] and train all plain/residual nets from scratch. We use SGD with a mini-batch size of 256. The learning rate starts from 0.1 and is divided by 10 when the error plateaus, and the models are trained for up to 60×10^4 iterations. We use a weight decay of 0.0001 and a momentum of 0.9. We do not use dropout [14], following the practice in [16].</p> <p>In testing, for comparison studies we adopt the standard 10-crop testing [21]. For best results, we adopt the fully-convolutional form as in [41, 13], and average the scores at multiple scales (images are resized such that the shorter side is in $\{224, 256, 384, 480, 640\}$).</p>	<p>Many of the tricks used in DL in this paragraph.</p>

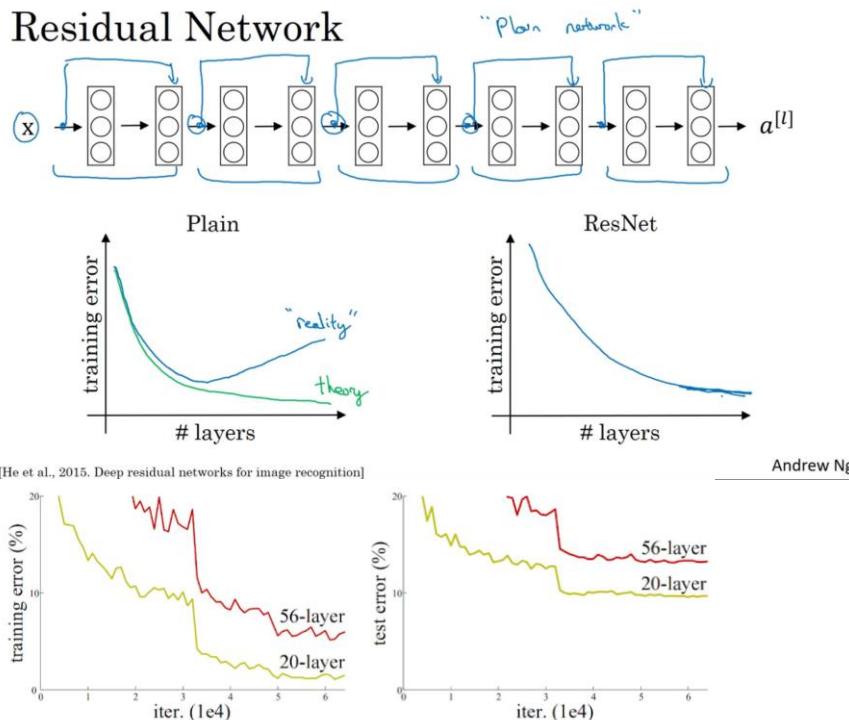
Residual block



[He et al., 2015. Deep residual networks for image recognition]

Andrew Ng

Residual Network

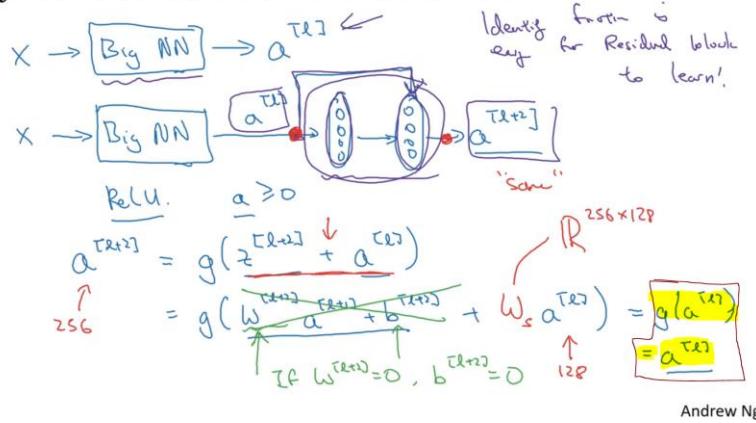


[He et al., 2015. Deep residual networks for image recognition]

Andrew Ng

Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer “plain” networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

Why do residual networks work?



Andrew Ng

If we consider that these two terms are 0 (due to weight decay -L2 regularization) then you are left with $g(a^l)$ which is equal to a^l because a^l is

Instead of having just the main path (the typical one) we take the a^l activation and propagate it further into the network as is. We add it to the activation of the destination layer before passing it through its nonlinearity.

ResNets typically use many of these residual blocks in sequence. They propagate activations deeper into the network.

You can't make a plain network very deep (you can't go easily to let's say 40+ layers) because due to vanishing/exploding gradients training becomes impossible so the training error starts to increase after a certain point when the number of layers increases a lot. This is referred to as the degradation problem (degradation of training accuracy as the number of layers increases).

With ResNets on the other hand, the action of moving an activation (an encoding) deeper into the network as is, allows to tackle the vanishing and exploding gradients problem in deep networks, and you can have networks with 40+ layers (100s or even 1000s)

(Ignore the red notes initially)
 Residual learning (ResNet) tackles this degradation problem by making it easy for the CNN to learn the identity function. I guess that this is important so that you have at least the same performance as before (the same learned encoding is just propagated further) without facing vanishing/exploding gradients. But we don't just want same performance, we want to make it better. And this might happen, because the extra layers might learn something useful that when added to the identity function produces a better encoding and the performance might be improved as a result. On the contrary, it is not easy for plain nets to learn the identity function by themselves.

the output of relu which means it already positive so passing through relu is simply the identity function.

The red notes is for the case in which the a^L and a^{L+2} have different dimensions. We just multiply a^L with a matrix W_S .

Alfredo's Lab

- With pooling you change the x and y dimensions of the encoding (the number of pixels)
 - With same convolution (1by1 or large), you can change the z dimension of the encoding (the number of channels) which would be equal to the number of kernels.

Input layer / samples

$$\begin{aligned}\mathcal{X} &= \{\mathbf{x}^{(i)} \in \mathbb{R}^n \mid \mathbf{x}^{(i)} \text{ is a data sample}\}_{i=1}^m \quad \text{input samples} \\ \mathcal{X} &= \{\mathbf{x}^{(i)} : \Omega \xrightarrow[\text{domain channels}]{} \mathbb{R}^c, \omega \mapsto \mathbf{x}^{(i)}(\omega)\}_{i=1}^m \\ \Omega &= \{1, 2, \dots, T/\Delta t\} \subset \mathbb{N}, \quad c \in \{1, 2, 5+1, \dots\} \\ \Omega &= \{1, \dots, h\} \times \{1, \dots, w\} \subset \mathbb{N}^2, \quad c \in \{1, 3, 20, \dots\} \\ \Omega &= \mathbb{R}^4 \times \mathbb{R}^4, \quad c = 1 \quad \mathbf{x}(\omega_1, \omega_2) = \begin{pmatrix} r(\omega_1, \omega_2) \\ g(\omega_1, \omega_2) \\ b(\omega_1, \omega_2) \end{pmatrix} \quad \begin{matrix} \text{four-momentum} \\ \text{space-time} \end{matrix} \quad \begin{matrix} \text{Hamiltonian} \\ \text{width} \end{matrix} \quad \begin{matrix} \text{grey scale} \\ \text{colour hyperspectral} \end{matrix}\end{aligned}$$

Spectrometer data can have many channels (a lot of information for each point). Hyperspectral images have 20 channels, wavelengths that we don't see.

Formal definition of input signals (input data of CNNs)

An example: The domain is the audio data. We get samples from it. Suppose that each sample has 2 channels (in case of stereo signal, one for each speaker). So you can think of it like this: You have a function x that gets a sample and outputs 2 values for that input (each value represents a channel). This function x is the signal. It describes with specific information a specific point (specific location) of its domain Ω .

A common sampling frequency for audio data is 44 kHz. We get 44k samples from 1 sec of data. Δt is the sampling interval.

The input data can have more than one channels. Apart from this, you can think of a CNN with 2 kernels that process a signal with one channel, as a process that converts a mono signal to a stereo one. You have 2 kernels. You go from mono type of signal (the input vector) to stereo type signal (two outputs, one for each kernel). The linear classifier after the CNN treats the output of all channels as one combined vector and uses it as its input.

Signals can be represented as vectors



$x = [x_1 \ x_2 \ x_3 \ \dots \ x_t \ \dots]^T$
 x_t are waveform heights

A close-up photograph of a young tabby cat lying down, looking directly at the camera with large, expressive eyes. The cat has distinct brown stripes on its light-colored fur. It appears to be resting on a soft, textured surface.

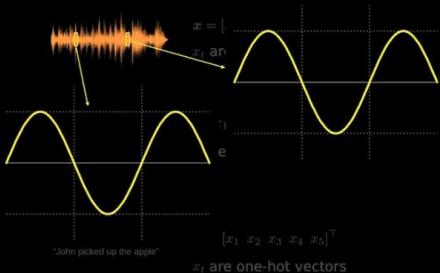
x_{ii} are pixel values

"John picked up the apple."

x_t are one-hot vector

In the 3rd case, each x is a 10k one hot encoding vector (if the language has 10k words). We can represent a sentence as a signal and process it with CNN? Where we have a signal with 10k channels?

Signals can be represented as vectors



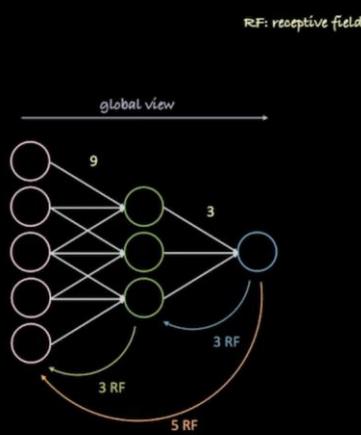
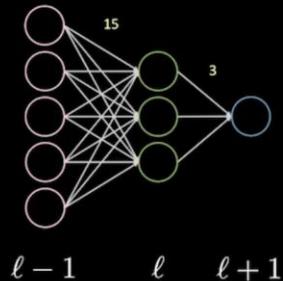
CNNs exploit **stationarity, locality** and **compositionality** of natural data

Stationarity: The same pattern happens many times within the input domain

Locality: two nearby pixels higher chance to have similar value

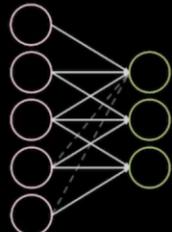
We exploit these properties of the data to spare computations (use of CNN instead of fully connected layer)

Locality \Rightarrow sparsity



A fully connected layer with 5 input units like this, has 15 connections between the first and second layer. The CNN has 9. It is sparser. Dropping connections means weights there are 0. So the convolution is a linear operation. **The input vector is multiplied by a matrix but it is a sparse matrix a matrix with a lot of 0s.** the other option would be to "slide" the kernel on the input and have multiple smaller matrix multiplications. **This later approach would require less memory but could be slower.** While the former requires more memory but might be faster (might be faster to multiply one large matrix in GPU than multiple smaller ones) In terms of CNN as space transformation, each unit focuses on a subspace of the whole space and transforms that subspace.

Stationarity \Rightarrow parameters sharing



- Parameters sharing**
- faster convergence
- better generalisation
- not constrained to input size
- kernel independence
⇒ high parallelisation
- Connection sparsity**
- reduced amount of computation



Faster convergence because each weight of the kernel gets multiple gradients in one step. So it has more info to move to the right direction.

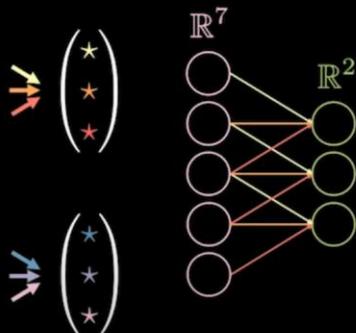
Not constrained to input size. You train a small CNN to detect faces and then slide it over a larger input. You can parallelize each kernel's computation. They are independent

Multi channels input (for example RGB images)

Kernels – 1D data

kernel size: $2 \times 7 \times 3$

1D data uses 3D kernels-collection!



The kernel size

2: number of kernels

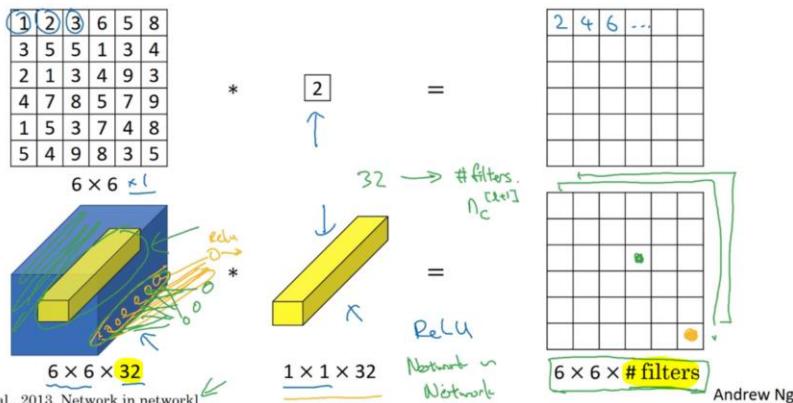
7: number of channels in the input

3: the receptive field

Suppose that your input has 7 channels (not just 3 like an RGB image). Think of the input vector on the surface of the screen, extended in the vertical direction 7 times. In this case you work like this: the kernel which has a receptive field of 3 units (yellow, orange, red is the first kernel) also extends in the vertical direction 7 times. So the yellow star of the kernel in reality it isn't just one weight, but seven weights, one for each channel. The output is still \mathbb{R}^2 because we use 2 kernels and we get one output for each kernel. So, the result of the convolutions of one kernel with each channel are added with each other and give one combined feature map. If we want to expand the input to higher dimensions we must use more kernels, so that each kernel has one output and put all together they form a high dimensional representation of the input.

As I understand it, the kernel looks at each channel with a different pattern because the weights can be different for each channel.

Why does a 1×1 convolution do?



A 1×1 convolution is a commonly used operation. It can be used to modify the number of channels, the 3rd dimension of the volume, because the resulting number of channels is the number of kernels used. Here in the picture we see one of the kernels (the yellow parallelogram). It is actually multiplied by the same volume of the blue volume, results in a value that then passes through a relu. This for every kernel. Even if the number of channels is kept the same the 1×1 convolution adds an extra non linearity so it might help with learning.

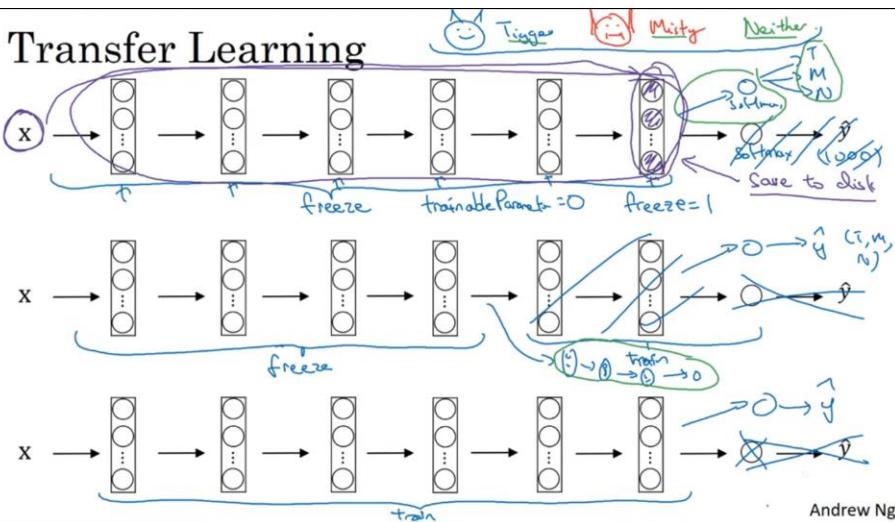
They are significantly cheaper than 3by3 convolutions (9 times less operations).

Misc

Transfer learning in CNNs

Most famous models are open source. They required large amounts of time and resources to be trained and finetuned for their hyperparameters. So, the most common way for making a project of your own, is to use an opensource model and fine tune it a bit just for your specific case.

Transfer Learning



If you have more training data available you can unfreeze some of the latest layers.

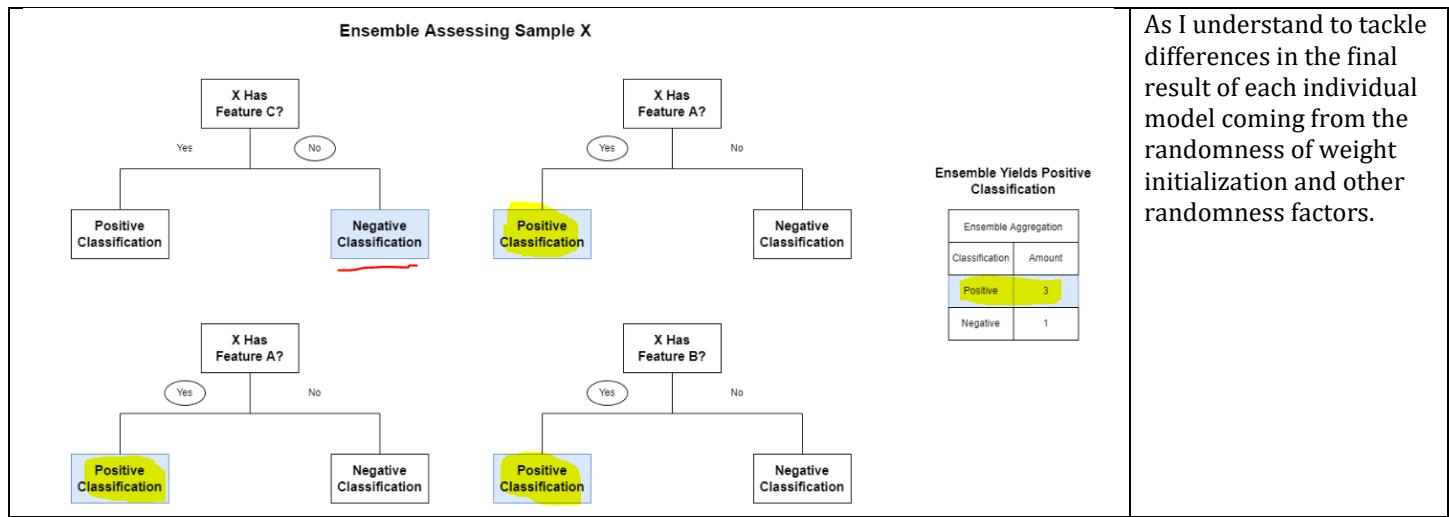
If you have a lot of training data you can unfreeze the whole network and use its weights only for weight initialization instead of starting with random weights.

For example, you want a CNN to classify your cats. You don't have much data for that task. You download a trained model and replace its softmax that classifies to 1000 classes with a new softmax for your 3 classes. You freeze all the layers and train the model with your limited data. In reality, you only train the softmax module.

Have in mind the save to disk approach. Since the whole network is frozen you can run it one time on your small training data set, just make inference, and save in the disk the output vector for each example. Then during the training process of your softmax, you don't have to make inference for each epoch, you just read the saved output for the given input. This makes the training faster.

Ensemble of models

Typically used when measuring the performance of a model

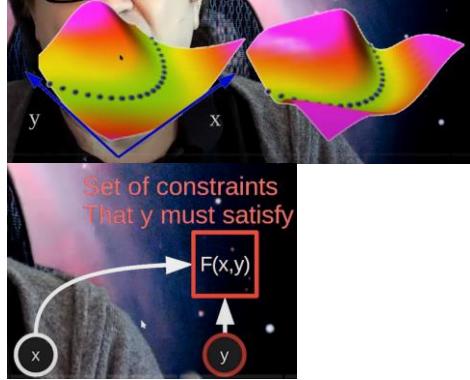


EBMs (Energy Based Models)

It is a formalization framework for describing a vast variety of architectures and techniques

You want to be able to represent uncertainty in the output, for example a given x can have multiple compatible y s, not just one. An EBM is a system (here the system $F(x, y)$) to which you give an x and a y and it determines if they are compatible or not (by outputting a scalar value, the so called energy, that measures their incompatibility). Energy based models is a formalization created by LeCun to describe a whole class of existing methods.

An energy function constructed during training. Blue dots are the data points, the data we observed.



During training you construct the energy function (the shape of the function). During inference, you are given an x and you try to find a y that minimizes the energy function for that given x , by running an optimization problem on the energy function you constructed during training. You descend on the energy function. The energy function is the cost function which you minimize during inference! Inference involves optimization (not SGD but GD)!

EBMs can be categorized in two main groups.

1. **Joint embedding architectures** and
2. **latent variable models.**

There are two ways to train them and construct the energy function.

1. **Contrastive**
The first one is contrastive methods where you push up the energy of specific points you know are unfeasible. But these methods are very data inefficient. You need a lot of data. In JEAs contrastive methods only work for cases in which the representations of the things you train on, are small.
The other way is composed of two categories, architectural and regularized methods where you don't have to select points to push their energy up. You just constraint the volume that has low energy. So they need less data for training, they are more efficient.
2. **Non contrastive** (architectural methods and regularized methods)

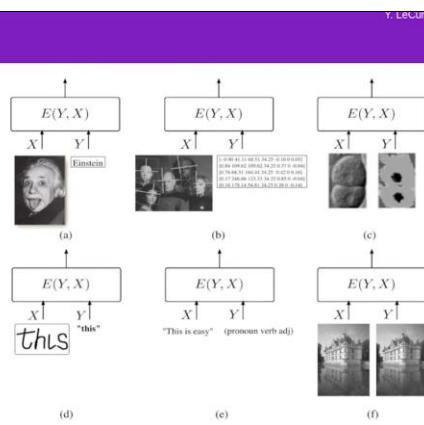
Regularized methods on JEAs is the most promising path (Lecun 2022)

In general you pretrain models in a self supervised manner (using EBMs, non-contrastive joint embedding models are very hot lately) so that they learn some good representations. Then you train these pretrained models in a supervised manner to fine tune the representations. And you need much less labeled data to achieve state of the art performance.

In these cases it is better to use EBMs than standard models.

When inference is hard

- ▶ Cases where inference is hard:
- ▶ Output is a high-dimensional object with structure:
 - ▶ Sequence, image, video,...
- ▶ Output has compositional structure:
 - ▶ Text, action sequence, ... ,
- ▶ Output results from a long chain of reasoning
 - ▶ That can be reduced to an optimization problem

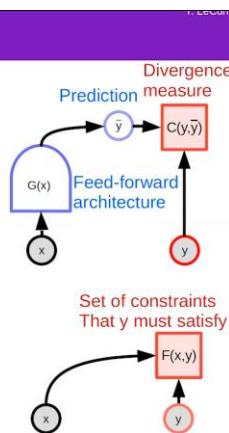


When the output is a continuous a high dimensional variable where there is uncertainty or it is compositional (structured prediction) then it might be better to use EBMs rather than directly predicting the output.

EBMs are Factor graphs

Energy-Based Models

- ▶ Feed-forward nets use a finite number of steps to produce a single output.
- ▶ What if...
- ▶ The problem requires a complex computation to produce its output? (complex inference)
- ▶ There are multiple possible outputs for a single input? (e.g. predicting future video frames)
- ▶ Inference through constraint satisfaction
 - ▶ Finding an output that satisfies constraints: e.g a linguistically correct translation or speech transcription.
 - ▶ Maximum likelihood inference in graphical models



You want a range of possible outputs, not just a single one.

We have to have a way to represent the uncertainty. Not the module G can't just be a function of x because that will give one y for each x . One solution is to have an energy module instead of the function G . It produces a single scalar value that denotes how compatible are x and y . Low values if compatible. High value if incompatible. That's why they are called energy models. In a physical system if the energy is high it wants to move to low energy (think of a spring, etc.). Here if we have high energy the x and y should change (the system needs to move).

EBMs are a type of graph networks (factor graphs). They are models where you have 2 kinds of nodes. Factor nodes (the rectangles) which produce a scalar output which says if its inputs are incompatible or not. Then you have variable nodes (the circles) which are the inputs to factor nodes.

Training

Contrastive Methods vs Regularized/Architectural Methods

Y. LeCun

- ▶ Contrastive: [they all are different ways to pick which points to push up]
 - ▶ C1: push down of the energy of data points, push up everywhere else: Max likelihood (needs tractable partition function or variational approximation)
 - ▶ C2: push down of the energy of data points, push up on chosen locations: max likelihood with MC/MMC/HMC, Contrastive divergence, Metric learning/Siamese nets, Ratio Matching, Noise Contrastive Estimation, Min Probability Flow, adversarial generator/GANs
 - ▶ C3: train a function that maps points off the data manifold to points on the data manifold: denoising auto-encoder, masked auto-encoder (e.g. BERT)
- ▶ Regularized/Architectural: [Different ways to limit the information capacity of the latent representation]
 - ▶ A1: build the machine so that the volume of low energy space is bounded: PCA, K-means, Gaussian Mixture Model, Square ICA, normalizing flows...
 - ▶ A2: use a regularization term that measures the volume of space that has low energy: Sparse coding, sparse auto-encoder, LISTA, Variational Auto-Encoders, discretization/VQ/VQVAE.
 - ▶ A3: $F(x, y) = C(y, G(x, y))$, make $G(x, y)$ as "constant" as possible with respect to y : Contracting auto-encoder, saturating auto-encoder
 - ▶ A4: minimize the gradient and maximize the curvature around data points: score matching

There are two ways to train an energy function. Contrastive and non-contrastive (regularized) methods.

In architectural non-contrastive methods the volume of zero energy is constrained by the architecture of the model (k-means).

In regularized methods, you use a regularization term in the energy function, so that you pay a price for increasing the volume of zero energy space. We do it by constraining the volume z can occupy. In reality what you end up doing, is to shrink the volume of low energy around the data manifold.

The energy function could represent probabilities or not. Usually we don't need it to represent probability. We are just fine by having it give us low energy for plausible values and high for the others. So, depending on what type of energy you want (probabilistic or not) you select the proper loss function for training.

- Probabilistic
 - For example max likelihood. It can't be applied in cases where the output is not discrete
- Contrastive
 - Hinge pair loss, NCE. Very popular. We don't care for the energy to be a probability distribution. The challenge here is to select the proper negative samples to push their energy up.

Loss functions for contrastive methods

<https://youtu.be/AOFUZZZ6KyU?t=89> see 3 minutes for an overview of the loss functions

1. Negative loss likelihood pulls up the energy everywhere
2. Sampling methods (Monte Carlo methods) you approximate negative loss likelihood with a set of points that you have to choose

3. Contrastive loss methods where the loss function is made of two energy parts, one part where you have the contribution from the energy of positive examples and one from the negative examples. The loss function should be an increasing function of the positive part and a decreasing function of the negative part so that it pushes the former down and the latter up, maybe up to a limit (for example hinge loss function)

<p><u>Compression.</u> Loss function for Contrastive Methods</p> <p><u>m</u> in contrastive loss function \rightarrow threshold of distance b/w low and high energy pairs (at least flat, it should be)</p> <p>You put two terms in the loss ^{here}, one term is the energy of the positive pair and the other one the energy of the negative term, you combine them in a way where the expression decreases decreases when the first decreases and the 2nd increases. This will push the energies of the pairs in the right direction. For example <u>hinge loss</u></p> <p><u>InfoNCE loss</u>: Two very popular last 3-4 years especially for training joint embedding systems. The concept is that you have only 1 positive pair within the batch and all others negative. We use logsoftmax on the negatives so the point with the ^{negative} smallest energy will be pushed up harder than the others (It will take most of the gradient)</p> <p><u>Example of joint embedding</u>: You show <u>image</u> in one and a <u>text</u> or a <u>query</u> to the other.</p> <p><u>WaveNet VITS</u></p>	<p>InfoNCE is used for joint embedding methods</p>
---	--

Inference

<ul style="list-style-type: none"> ► Energy function $F(x,y)$ scalar-valued. ► Takes low values when y is compatible with x and higher values when y is less compatible with x. ► Inference: find values of y that make $F(x,y)$ small. ► There may be multiple solutions $\hat{y} = \arg\min_y F(x,y)$ ► Note: the energy is used for inference, not for learning ► Example ► Blue dots are data points 	<p>Notice during inference y is also an input along with the observation x. and the actual y that the model infers, is the result of an optimization problem, y-optimum (minimize energy w.r.t y).</p>
---	---

inference is not just passing x through the model. You pass an x and the y is the one that minimizes the energy function. In EBMs inference contains optimization. So, the energy is used for inference. Not SGD, but typical Gradient descent. During inference both x and y are inputs to the model and you get an energy value for them describing their incompatibility. You do GD (W.r.t y and also z in latent variable models) in order to find a y that minimizes energy. Y_{check} is the notation for the y that minimizes energy for a given x .

Conditional and unconditional EBMs

▶ Conditional EBM: $F(x,y)$
 ▶ Unconditional EBM: $F(y)$
 ▶ measures the compatibility between the components of y
 ▶ If we don't know in advance which part of y is known and which part is unknown?

Self-Supervised Learning

- Predict any part of the input from any other part.
- Predict the future from the past.
- Predict the future from the recent past.
- Predict the past from the present.
- Predict the top from the bottom.
- Predict the occluded from the visible
- Pretend there is a part of the input you don't know and predict that.

If you imagine x and y as a one signal, in conditional EBM you know what part of it you will observe. This part is the x . y might exist in the training set but not in the test set.

In unconditional you don't know what part you will observe. So the whole signal is y . (and you want to see if this y is compatible with what? I guess if it is plausible) PCA, K-means can be framed as unconditional EBM.



Probability distributions as energy

You can transform an energy function to a probability distribution but this is not always possible or desirable. (you use the Gibbs-Boltzmann distribution. Softmax is an instance of a Gibbs-Boltzmann distribution). There are cases where the space of y is so large that you can't calculate the integral of all possible y s (that should add up to 1). This is a reason why EBMs are used. You forget about probabilities (make the most probable prediction) and you just care about the prediction being of low energy.

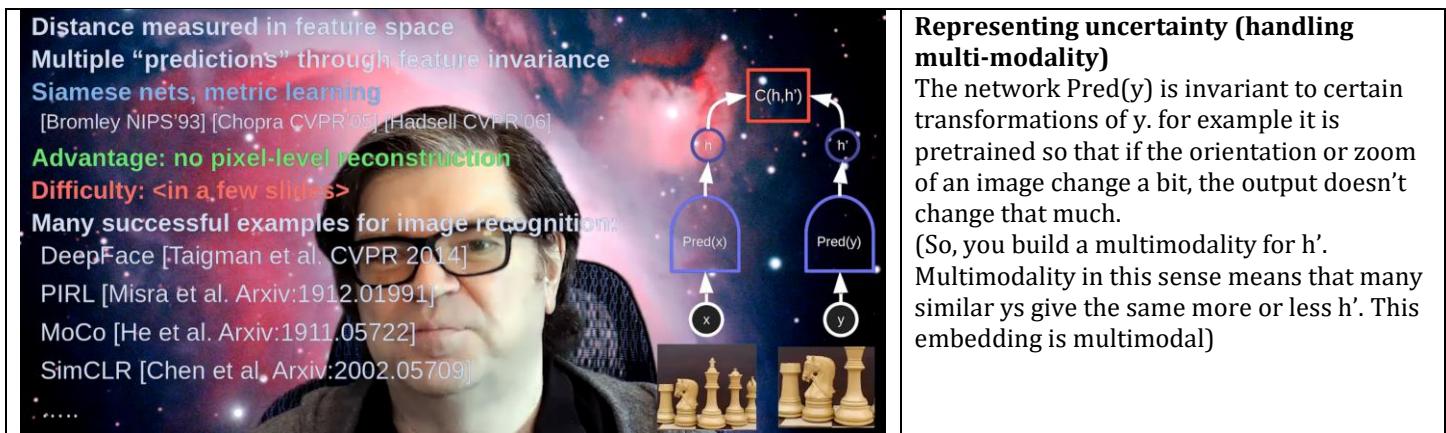
EBM Architectures

These are the two big families of EBMs. They are architectures for the module $F(x, y)$

- Joint embedding architectures
- Latent variable models

Joint embedding architectures JEAs

Popular from 2020 for image recognition without labeled data. They work very well for that. But they require huge amounts of resources for training because they need a lot of negative examples to push their energy up.



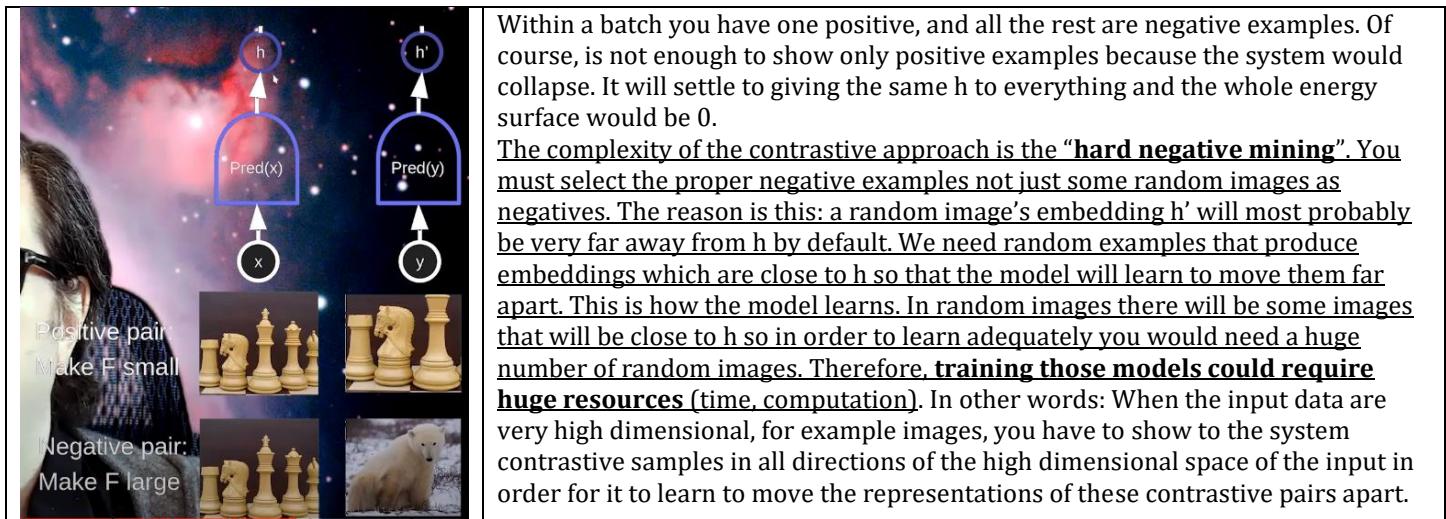
So if you use this trained network $\text{Pred}(y)$ in this system, and pass an x to the system, the y s that produce low energy would be images of x but with a bit different orientation or zoom.

Inference

Energy function creation/training

➤ Contrastive learning

This method is very popular in 2020 onwards. The examples for which the energy should be pushed up (the negative samples) are pictures picked by us. In general, contrastive methods are very expensive due to the hard negative mining problem. They only work for cases in which you can generate easily negative samples and the embedding space in which you make the comparison is low dimensional enough so that you don't have to create negative examples in a huge number of dimensions.



For example, if you wanted to train the SimCLR system which is made of two CNNs, you would need so many examples of image pairs that if you were to use AWS you would have to pay \$5m! so contrastive methods are very expensive. There is a huge incentive to find out alternative methods. MOCO is such a try.

There are various non contrastive training methods developed during the last years. Methods that don't require negative samples. For example SwAV, Barlow Twins (see models section).

➤ Non contrastive learning

Clustering, distillation, redundancy reduction

<https://www.youtube.com/watch?v=8L10w1KoOU8&list=PLLHTzKZzVU9e6xUfG10TkTWApKSZCzuBI&index=21>

Computer vision with Joint Embedding Architectures (JEAs)

Joint embedding architectures are used extensively in computer vision for learning good visual representations in SSL setting. The general concept is this: learn to represent an image and various variants of it with the same vector. To avoid energy collapse you must apply some tricks. Each of these models use such a trick.

In general video prediction is an unsolved problem as of 2021.

There are 2 ways to evaluate the learned representations:

1. Freeze the representations and use them for linear classification in known datasets
2. Use them as weight initializations and fine tune them for a specific task with a labeled dataset

<h2>Many ways to avoid trivial solutions</h2> <p>Similarity Maximization Objective</p> <ul style="list-style-type: none">• Contrastive learning<ul style="list-style-type: none">• MoCo, PIRL, SimCLR• Clustering<ul style="list-style-type: none">• DeepCluster, SeLA, SwAV• Distillation<ul style="list-style-type: none">• BYOL, SimSiam <p>Redundancy Reduction Objective</p> <ul style="list-style-type: none">• Redundancy Reduction<ul style="list-style-type: none">• Barlow Twins	These are ways with which we try to avoid joint embedding models energy collapse (called trivial solution alternatively) where the model produces the same embedding for everything.
--	--

The best way to create good representations is to make them invariant to data augmentation.

Have in mind the example of jigsaw permutation, where the layers closest to the output were worse in relation to the previous ones. The representations of the latest layer were becoming specialized for the specific task at hand and weren't that good when evaluating them.

Contrastive methods notes

We avoid energy collapse using negative examples. There are specific ways to generate positive and negative examples for images, video, video + audio.

Ways for negative examples: 1. Each batch to different GPU 2. Memory bank 3. Momentum (good explanation at 37:30)

Clustering methods notes

We avoid energy collapse by setting a specific number of points where the energy should be 0. For example k-means. Notice that we usually use alternatives to k-means, for example some optimal transport algorithm like Sinkhorn-Knopp which forces identical number of entries in each cluster. Have also in mind soft clustering. An embedding can belong to more than one clusters with a weight to each one (0.8 to cluster 1, 0.2 to cluster 2). This helps to avoid the problem that you would have with an equal number of embeddings to each cluster in cases with cluster imbalances.

Audio + visual representations. They are useful for separating between frames that are similar visually (dancing/exercising) but have different audio (music in dancing).

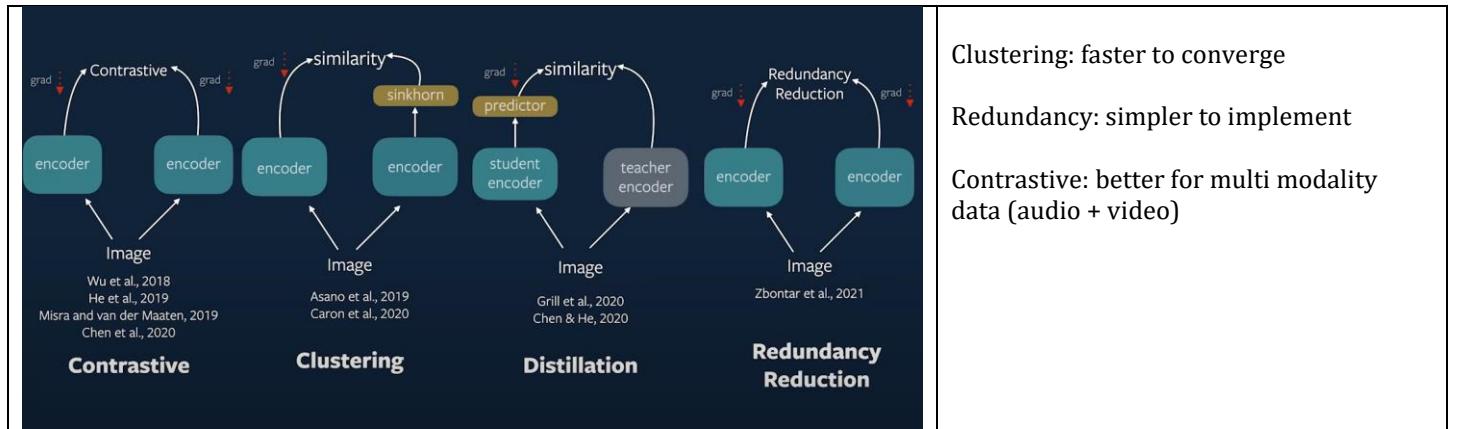
Distillation notes

You avoid collapse by introducing asymmetries between the two branches of the model. Asymmetries to the architecture or to the learning rule.

Redundancy reduction notes

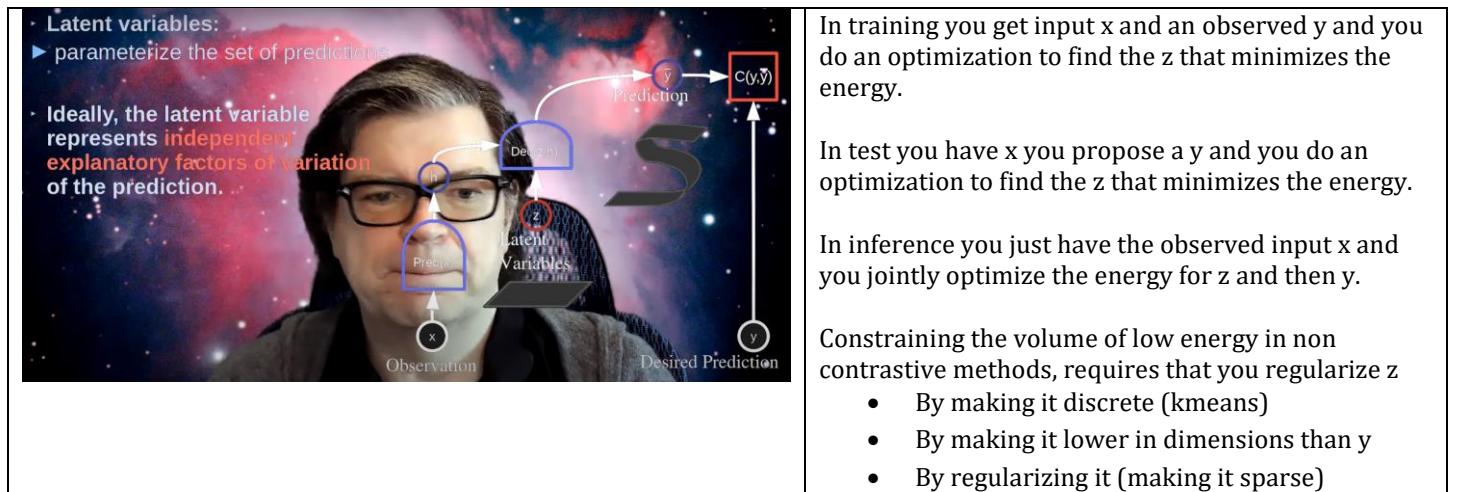
We avoid energy collapse by enforcing embeddings of different pictures to be uncorrelated with each other. First we have to force the neurons of an embedding to have no redundancy between them. Each used neuron should represent one feature.

Pros and cons of each method



Latent variable models LV-EBMs

Notes



In kmeans you force z to only take k values, so y-tilda can only take k values, so the y energy space can only have k points of zero energy.

Representing uncertainty (handling multi-modality)

You represent uncertainty with a latent variable. Latent variable: A variable that is not observed or given. It is an Internal variable of your model. You get x input, but the y_{hat} depends also on z . As you vary z over the surface, y_{hat} varies over the surface.

You want multiple y_{bars} for a given x (y_{bars} that lie on the S surface). This is the set of plausible y s for that x . You parametrize that surface with a latent variable z . You can allow it to vary along a surface, here a 2d surface meaning that z is 2d in this case. As z varies along the flat surface, y_{bar} varies along the S surface.

Important: z should not have that much information capacity, it should be low dimensional in relation to y . otherwise you might get energy collapse where the energy function is 0 everywhere. There are various ways to achieve this, for example regularizing the capacity of z (does this have anything to do with regularizing training methods meaning you only train latent variable models with regularized methods?). The rule is: **limiting the capacity of z , limits the volume of energy that can have low values.**

Ideally the latent variable represents independent explanatory factors of variation of the prediction. For example, if you want to read a handwritten word, it helps a lot if you know where the characters are split with each other. The position of the split can be the latent variable. Or if you have a 3d file of a face as x , and you want to see if the picture of a face (y) is from the same person you the 3d scan of, you have to rotate the 3d face to match the view of the picture. This rotation can be the latent variable.

Notice: dimensionality of z

In this case z was a scalar value, essentially 1 dimensional (z varies along a line), which means that the zero energy space will also be 1 dimensional. It will be a 1-dimensional subspace of y which is 2d in this case (y_1, y_2). The zero-energy space is a 1-dimensional subspace of that (a line that forms an ellipse). If z was 2d in this case, then it would be able to reach all locations of the y space (the ambient space) and would give 0 energy to the whole y space, making the energy function flat. This is why z should be less powerful than y . It must be lower in dimensions or if it has more dimensions it should be sparse (for example in k-means z has the k dimensions which might be more than the dimensions of y , but it is one hot encoded -very sparse). It is a challenge to find a good size for the latent variable. It is a challenging research problem as of 2021.

In general, if you discretize z , like in k-means which can take k values, then you constraint the volume of low energy space. It will have zero energy in k locations.

If z varies over an n space, then y_{bar} would vary at most, over an n -space. It could vary over an less than n space because the neural net would map many points of z onto the same y_{bar} (reduced the dimensions). So limiting the dimensions of z , in general, limits the dimensions of space that can have low energy (the space of y_{bar}).

Q: I would assume that as a 1d z corresponds to a 1d zero-energy space, a 2d z would correspond to a 2d zero-energy space... But at 1:02:44 you said that if z is 2d, there is no known constraint on the zero-energy space and that the zero-energy space would be flat. Is there an intuition on why is that? Or maybe I misunderstood?

A: If the ambient space is 2d, a 2d latent would let you reach all locations, effectively giving you a flat free energy, unless you limit the volume of low energy by using a regulariser.

Q: Ah I see... If I understand correctly, the ambient space is the y space which is 2d in this case (y_1, y_2). And this would imply that in other cases, z can be high dimensional so long as it has less dimensions than y (the ambient space), so that the zero energy space is a subspace of the ambient space.

A: Doesn't necessarily need to be of smaller dimensionality. For example it could be larger but sparse.

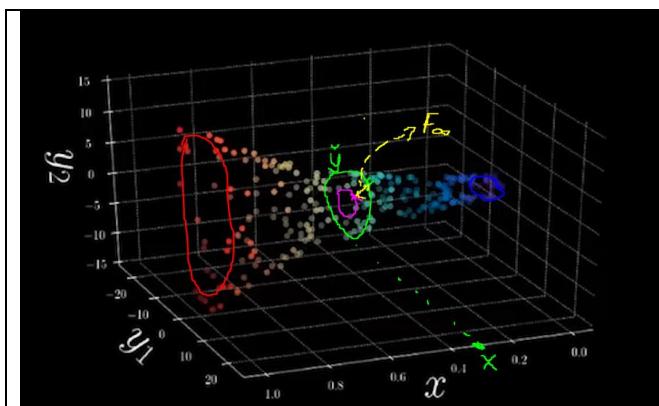
Inference

You give it an x and a proposed y . The system needs to choose a \bar{y} that minimizes the energy. A way to do this, is to pick a z value that produces a \bar{y} that minimizes energy. So to pick the best value for the latent variable. So during inference, you make an optimization w.r.t the latent variable z . what does this mean: You give it an observation x and a proposed prediction y . the system finds internally the closest plausible \bar{y} to that given y (the point on the s surface that is the closest to y). the output of the system (the energy that it gives to the pair x, y) is the energy (incompatibility) of this \bar{y} with the proposed y .

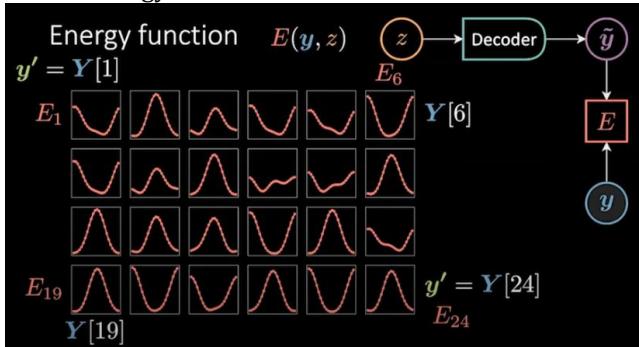
That's the first part. But what you want from your system is for it to propose an y , the most plausible y (which would probably be different than the proposed one). So you have to also optimize the energy w.r.t variable y too. and then output that optimized y that minimizes the energy function for that given x (and given z).

So first you find the best value for the variable z , given x and y , and then you take that z value for granted, and find the best value of y for that given x (and z). This is simultaneous minimization of energy w.r.t z and y (Free energy). ... How you minimize the energy simultaneously for z and y . You define a new model that encapsulates z . you minimize over z in that internal model. Then you have a typical EBM without z and you minimize over y . The formula of the energy of that model is something that physicists call free energy.

A specific example



The 24 energy functions.



The data set is this manifold.

Let's focus on one slice of it. One given x has many plausible y s (the points of the ellipse), that's why we need to use energy methods. For a given x , the plausible values of y form an ellipse (the green one). It is composed of 24 datapoints.

We use a latent variable in our model. For a given x , our model will produce a y . the y will vary with z .

Suppose we have a trained model. For a given x , as we change z , the y moves along the purple ellipse. (this model is not so well trained because the purple ellipse is not on the green ellipse)

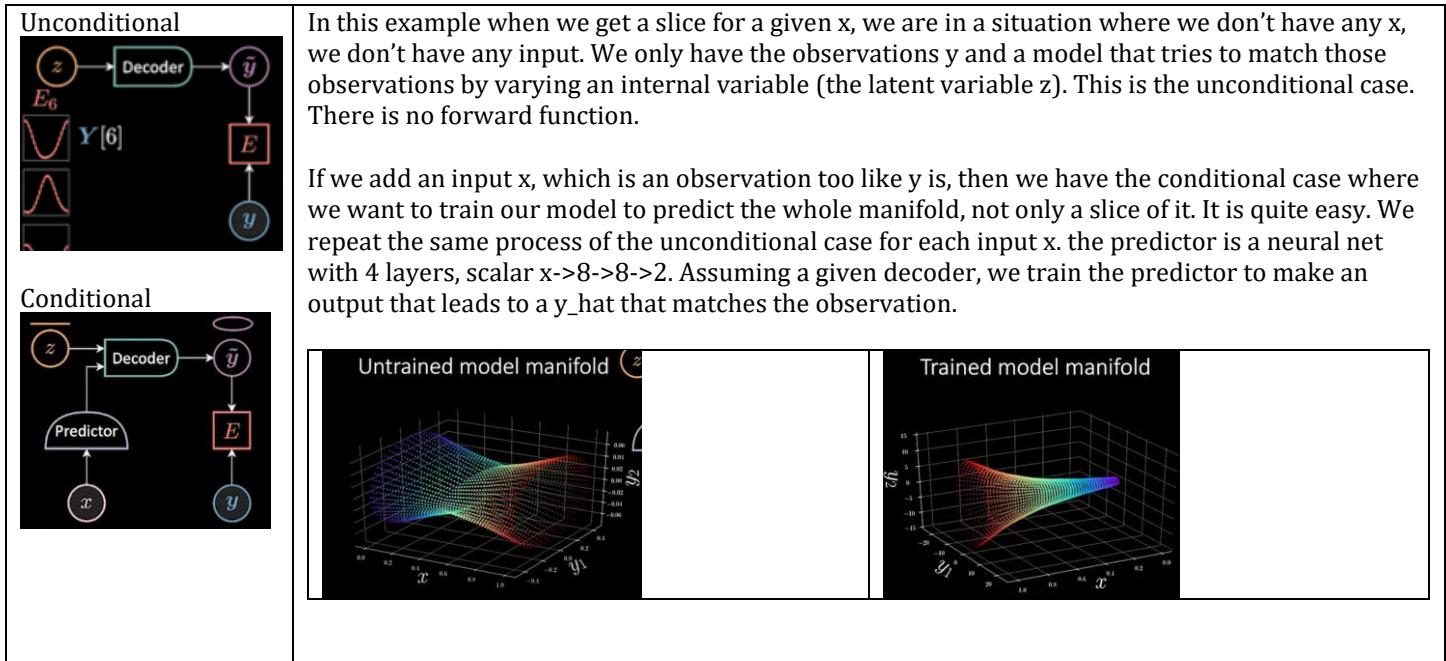
The energy is the distance (square root of distance so that it is always a positive number) of the predicted y with a given datapoint.

The predicted y varies with z , so the energy function of the first datapoint varies with z . this function has a minimum. It corresponds with a z_{check} value (a z value that gives the lowest energy for that datapoint and that x) that gives the prediction which is closer to the given datapoint. **This energy of the z_{check} is the free energy F for that datapoint** (F is minimization of E over z)

We have 24 energy functions like this, one for each datapoint.

So, in inference, we give a x and a proposed y (which could be one of the datapoints) and the model will predict another y (\bar{y}) which corresponds to the free energy (corresponds to the z value that gives a \bar{y} that has the smallest distance with the proposed y). the model will also output this energy value.

Notice: Conditional and unconditional case



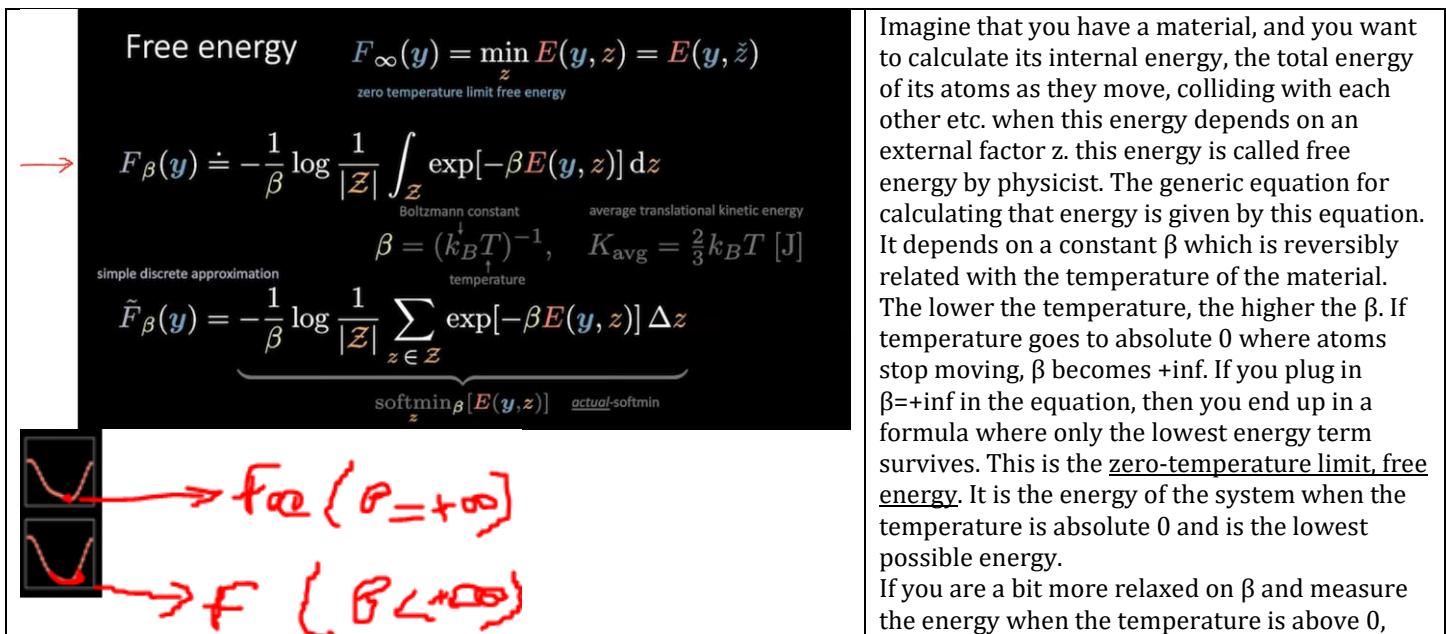
Notice

- If the decoder is also a function that needs to be learned, then things change. Learning good decoders is a research topic.
- Another even more challenging research topic is to find a proper non scalar z, to find the dimensionality of z.

Free energy

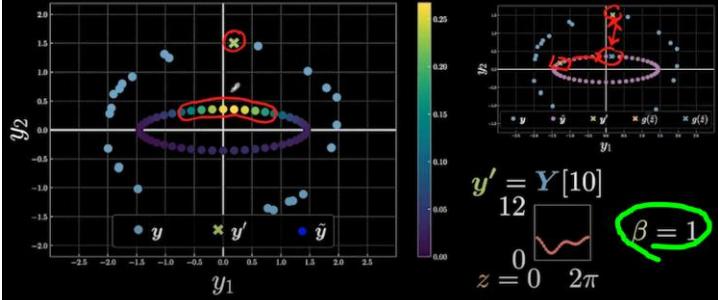
<https://youtu.be/XIMaWj5Yj0Q?t=796> a good 2 minutes explanation by Alfredo to gain intuition about free energy in physics and also in latent variable EBMs.

There is a more general term for the free energy that contains the β term (inverse temperature) and kind of controls how strict we want to be in selecting z-check (in selecting the free energy of a given observation). Instead of selecting the z that gives the best $y\tilde{}$ with the lowest energy, we select a range of possible zs and calculate the average energy of those (marginalization instead of minimization). I think that this generic term for free energy is derived from the VAEs where we replace the free energy with a variational approximation of it. I guess that this general term can be used by all latent variable models so that we can create a smoother (convex) energy function that can be used for gradient based optimization methods.





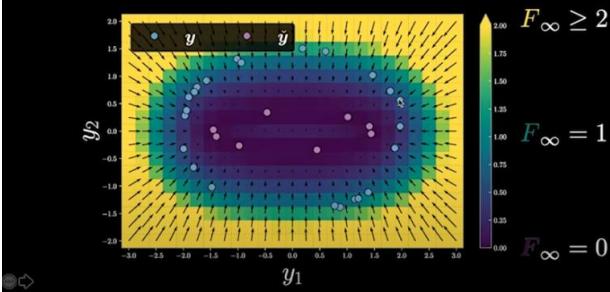
$$\text{Free energy } F_\beta(\mathbf{y}) \doteq -\frac{1}{\beta} \log \frac{1}{|\mathcal{Z}|} \int_{\mathcal{Z}} \exp[-\beta E(\mathbf{y}, z)] dz$$



then the free energy now is the average of all energies of the zs with the lowest energy. When the temperature becomes infinite, β becomes 0 and the free energy is now the average of the energies of all zs. Essentially at this point, you have eliminated the effect of the latent variable z because its value makes no difference now. The energy is always the average of all predictions (one prediction for each z).

Having the temperature to 0, might cause overfitting because you are too strict on the prediction.

$$\text{Free energy } F_\infty(\mathbf{y}) = \min_z E(\mathbf{y}, z) = E(\mathbf{y}, \check{z})$$

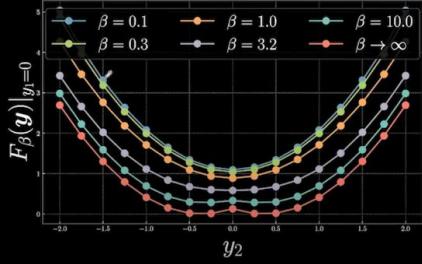


This is the zero temp free energy for this example. we pick one specific z that gives the minimum energy. This heatmap shows this energy for all points of the domain. For example a specific $y(y_1, y_2)$ has one specific energy value. Here we see the energy of all y s.

Notice that we have a pick in the center of the ellipse. This is not ideal because we will do an optimization on the energy surface (for inference when finding the best y for given x and z ?) and this might lead the optimization to local minima. We want a convex energy surface.

A solution is to increase the “temperature” or in other words, being more relaxed when choosing the free energy of a particular y taking the average of the closest zs. This will give a larger but smoother energy function.

$$\text{Free energy } F_\beta(\mathbf{y}) \doteq -\frac{1}{\beta} \log \frac{1}{|\mathcal{Z}|} \int_{\mathcal{Z}} \exp[-\beta E(\mathbf{y}, z)] dz$$



Energy function creation/training

Finding a well-designed energy function which is parameterized by the parameters of the model.

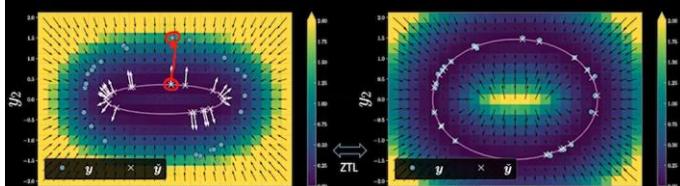
Loss functional

$$\mathcal{L}(F(\cdot), \mathbf{Y}) = \frac{1}{N} \sum_{n=1}^N \ell(F(\cdot), \mathbf{y}^{(n)}) \in \mathbb{R}$$

$$\ell_{\text{energy}}(F(\cdot), \mathbf{y}) = F(\mathbf{y})$$

$$\ell_{\text{hinge}}(F(\cdot), \mathbf{y}, \hat{\mathbf{y}}) = (m - [F(\hat{\mathbf{y}}) - F(\mathbf{y})])^+$$

$$\ell_{\log}(F(\cdot), \mathbf{y}, \hat{\mathbf{y}}) = \log(1 + \exp[F(\hat{\mathbf{y}}) - F(\mathbf{y})])$$



Common loss functions (loss functionals more formally).

A functional is something that gives you a scalar value given a function. You give it a function and it gives you a scalar value. It is a scoring mechanism for a function. In the context of energy functions, the functional will tell you how good or bad the energy function is.

You have some initial parameters that give you a prediction that is not good, like the purple ellipse. In training, you get an x , you predict a y by taking the free energy by minimizing the energy (square distance) with respect to z and then you push that predicted y by the gradient (the gradient of the square distance is the distance) so you push the predicted points to the datapoints. This is what happens in training. Of course you have to select a proper loss function.

Compression with latent variable EBMs

You have a frame of a video. You want to be able to predict what happens next. The only way to deal with this uncertainty on the infinite number of outcomes, is to use latent variables. There are a lot of things within the video that can't be exactly predicted, they are random, like the position of a falling pencil, and should be parameterized with latent variables. Then you just transmit a frame and the latent variables. The receiver can reconstruct the video (or at least some frames as I understand) from the latent variables. There are various startups in 2021 that work on this. Traditional approach is similar but the "latent variables" are designed not learned.

Regularizing

These are methods for which we restrict the volume of low energy by regularizing the latent variable. Finding the best ways to limit the information capacity of a latent variable in predictive models like these, is a main area of focus of LeCun's team. It is not a solved problem in 2021 and it is a problem that needs to be solved for progress in SSL (predicting video etc.). There are cases in which current methods work, but for specific cases.

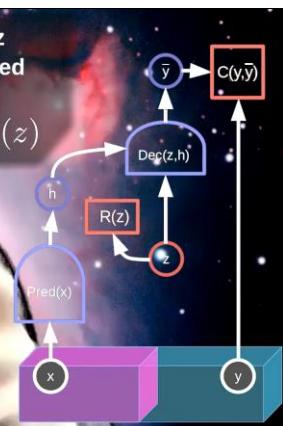
Regularized training methods for LV-EBMs

- Regularizer $R(z)$ limits the information capacity of z
- Without regularization, every y may be reconstructed exactly (flat energy surface)

$$E(x, y, z) = C(y, \text{Dec}(\text{Pred}(x), z)) + \lambda R(z)$$

- Examples of $R(z)$:

- Effective dimension
- Quantization / discretization
- L0 norm (# of non-0 components)
- L1 norm with decoder normalization
- Maximize lateral inhibition / competition
- Add noise to z while limiting its L2 norm (VAE)



The concept is that we add a regularization term in the cost. This has the effect of making the z vector sparse. Notice that as of 2021, this method works only for linear decoders.

Sparse coding and Target prop (Sparse Autoencoder)

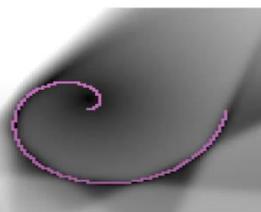
https://youtu.be/bdebHVF_mo?t=2018

Limiting the capacity of z with a L1 norm minimization, on a linear decoder ($\bar{y} = W^*z$, there is no relu or any other nonlinearity) is called sparse modelling. And learning that linear decoder is called sparse coding. In sparse coding the regularization term $R(z)$ is the L1 norm of z . This forces z to have a small number of non zero attributes. This has the effect that any output will be a linear combination of a small number of z attributes. It is important to note that the columns of W matrix must be normalized so that the square norm of each column is 1 or some other constant (the sum of the squares of the attributes of a column). This way you force the weights to be small, because normally they will become large during training so that z becomes close to zero (since the regularization term tries to decrease z and w^*z product should be close to y).

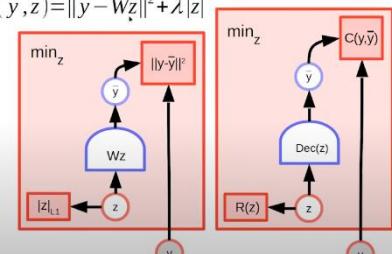
But inference on sparse models can be very expensive (it has this ISTA optimization). So if the inference is extremely inefficient you can use the target-prop technique that replaces the ISTA optimization with an approximate solution produced by a neural network and is actually faster (Because the decoder produces a good approximation of z check and you only need then a few iterations of ISTA). So you pay a price by learning that encoder, but then you have a more efficient inference.

Unconditional Regularized Latent Variable EBM

- ▶ Unconditional form. Reconstruction (no x , no predictor).
- ▶ Example: sparse coding / sparse modeling
- ▶ Linear decoder
- ▶ L1 regularizer on Z



$E(y, z) = \|y - Wz\|^2 + \lambda |z|$



Sparse Modeling [Olshausen & Field 1997]

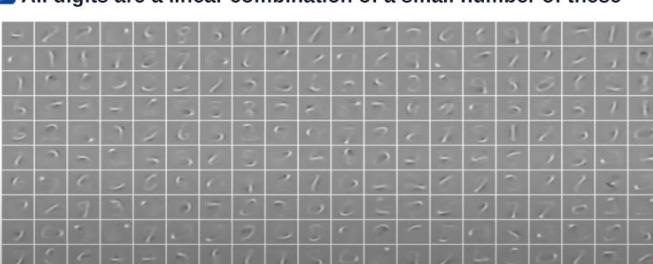
- ▶ Energy function $E(y, z) = \|y - wz\|^2 + \alpha \sum_j |z_j|$
- ▶ Capacity of latent variable z limited by L1 norm (sparsity)
- ▶ Columns of dictionary matrix w are normalized

▶ Procedure: repeat:

1. pick a training sample y
2. inference: find the optimal z $\hat{z} = \operatorname{argmin}_z E(y, z)$
 - Use the ISTA algorithm $z(t+1) = \text{Shrink}_{\alpha\eta} [z(t) - \eta w^t (wz(t) - y)]$
3. Update W $w \leftarrow w - \eta \partial E(y, \hat{z}) / \partial w = w + \eta z(y - wz)^t$
4. Normalize columns of W to a constant (e.g. 1).

Sparse Modeling on handwritten digits (MNIST)

- Basis functions (columns of decoder matrix) are digit parts
- All digits are a linear combination of a small number of these



If you train a system with sparse coding on MNIST, then these are the columns (components) of the matrix W that the system learns. It means that any digit can be generated by a linear combination of a small number of these components. This is the result of sparse coding. You make it so that you have a large enough number of components, from which you only need a small number of them for creating any output.

Sparse coding comes down to constraining W to have a limited L1 norm. you normalize its components so that they do not become too large. The problem with non-linear decoders is that we don't know as of 2021 how to do this normalization.

Sparse coding is a Late 90s proposal, originated from neuroscientists.

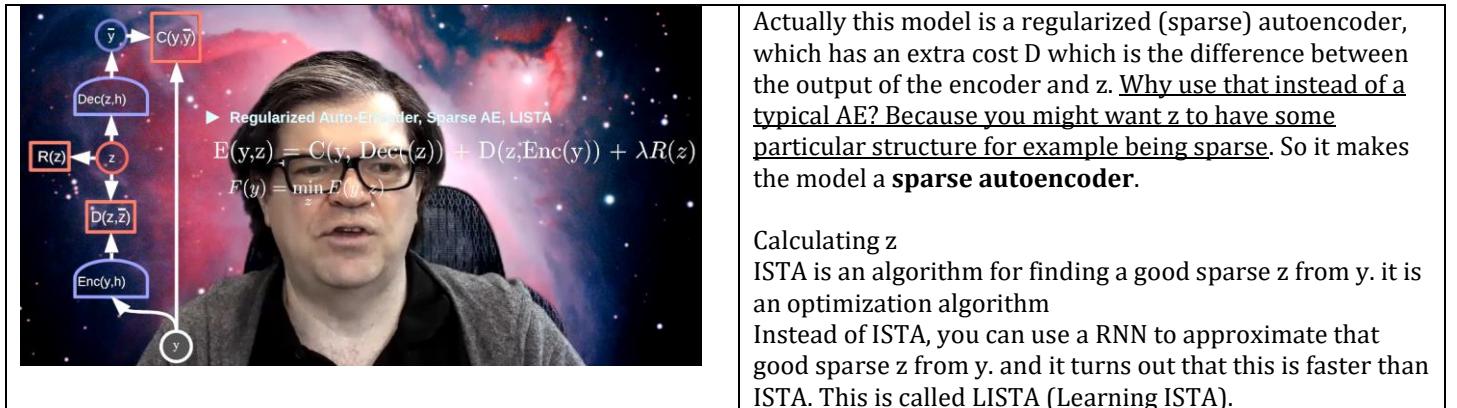
ISTA is an algorithm for finding a good sparse z from y . it is an optimization algorithm

Target prop (Amortized inference or Sparse Autoencoder)

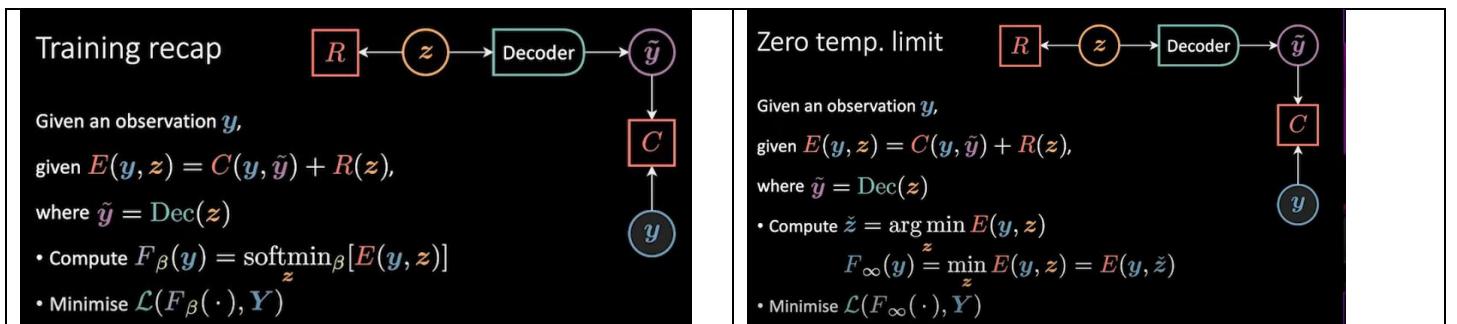
<https://youtu.be/AOFUZZZ6KyU?t=4339>

Amortized inference describes a type of network, that resembles autoencoders (it has an extra cost D that autoencoders don't have). Instead of finding the best z for given y with minimization, we use a neural network that learns to predict that optimal z from y . This technique is called amortized inference. You put a decoder between y and z . the decoder is the network that performs amortized inference. It creates a z -tilde which tries to become the z -check (the z with the smallest energy). So the encoder will learn how to approximate an optimization.

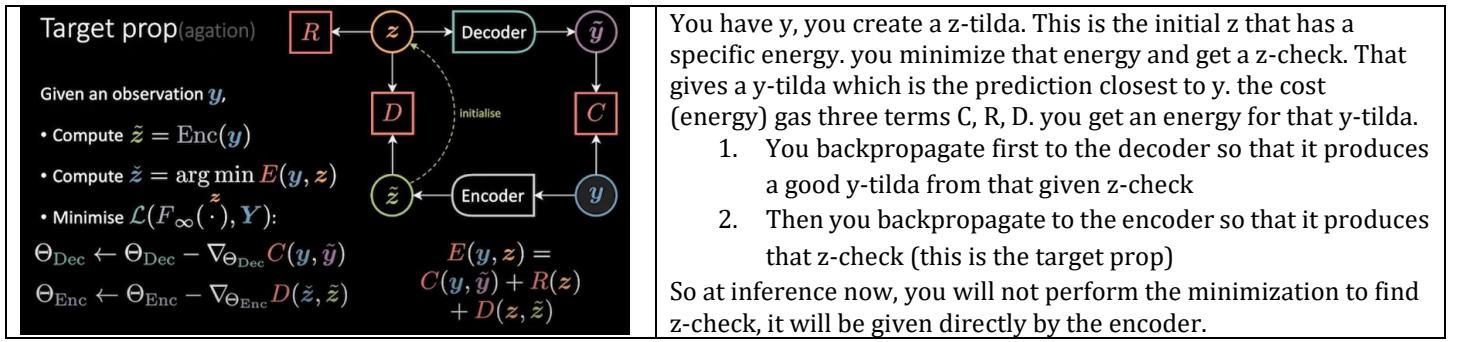
It is called by some target prop, because as you train the model to minimize the energy with respect to z, essentially you train to learn a good target for the encoder. The target is to produce z_check.



Typical latent variable model



Latent variable with amortized inference



Convolutional Sparse coding

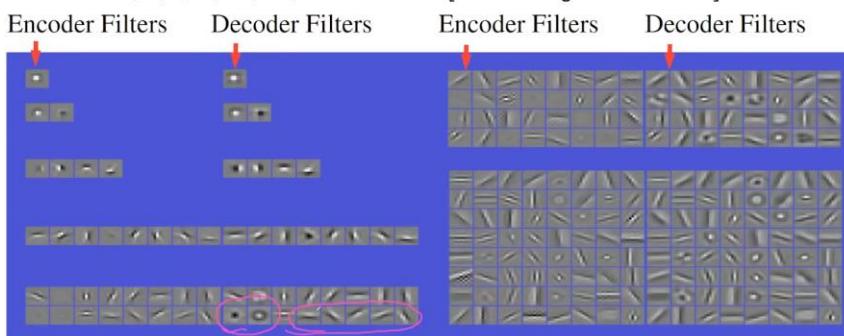
https://youtu.be/bdebHVF_mo?t=3110

You can use this sparse autoencoder (target prop) system to learn good representations of visual data (images). It actually learns some very good features similar to those with training a supervised CNN on MNIST. And is actually a very simple model. The decoder has only 2 conv layers. So we can use the layer of the encoder as a pretrained first layer for a CNN that we will fine tune for a specific task. As I understand the sparse AE can be used for creating a pretrained **first** CNN layer because as I understand the features they learn are the most basic ones. Maybe due to the fact that they need to be linearly combined (a few of them) to produce the output.

Convolutional Sparse Auto-Encoder on Natural Images

► Encoder filters and decoder filters. Decoder is linear (convolutional)

► with 1, 2, 4, 8, 16, 32, and 64 filters [Kavukcuoglu NIPS 2010]



Paper 2011

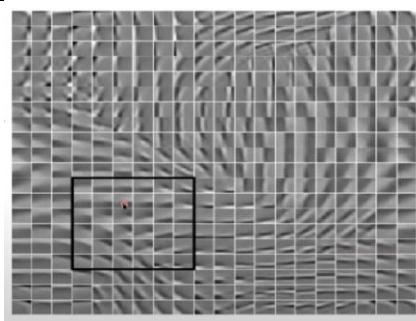
Y is an image that you try to reconstruct. The linear decoder is now a convolutional layer. You can learn very nice features just with SSL without labeled data, just by reconstructing the input image. The architecture is quite simple. You use this technique to pretrain a neural net.

Reconstructing each layer of a CNN

Notice another SSL technique for learning good features: You pretrain each individual layer of a CNN as a sparse autoencoder as described above. You train each layer to produce embeddings (features) that can reconstruct the input. You train the first layer like that. Then you fix it, you drop the decoder and use the decoder as the first layer of the CNN. Its output is input to the next layer. Then you train the second layer as a sparse autoencoder and so on. This idea was mentioned by Hinton too as a technique that gives good results. You can do the same with DAE or VAE.

Group sparsity

It's a very similar technique where z is now regularized with its L2 norm. You split components of z in groups. You get the L2 norm of each group. Then you add those terms. The sum is the regularization term. It forces only a few groups to be active. (within a group there is no sparsity). You can overlap groups (this gives the continuity effect shown below). The fact that only some groups can be active, forces the model to put features that appear together to the same group.



This produces this kind of features where they are continuous. Also features that appear together in the input are grouped together (they are represented by similar z).
This continuity has been seen in the cortex of some animals (not all)

Regularizing through temporal consistency

https://youtu.be/bdebHVF_mo?t=4128

This is another regularization method.

The inputs y are functions of time, they change with time. They can be video frames and you want to be able to predict the next frame based on the previous two. Or you might get an image as $y(t-2)$ then translate it and rotate it a bit, that's $y(t-1)$, repeat again for $y(t)$. You want to predict $y(t+1)$ which is further translation and rotation of the image.

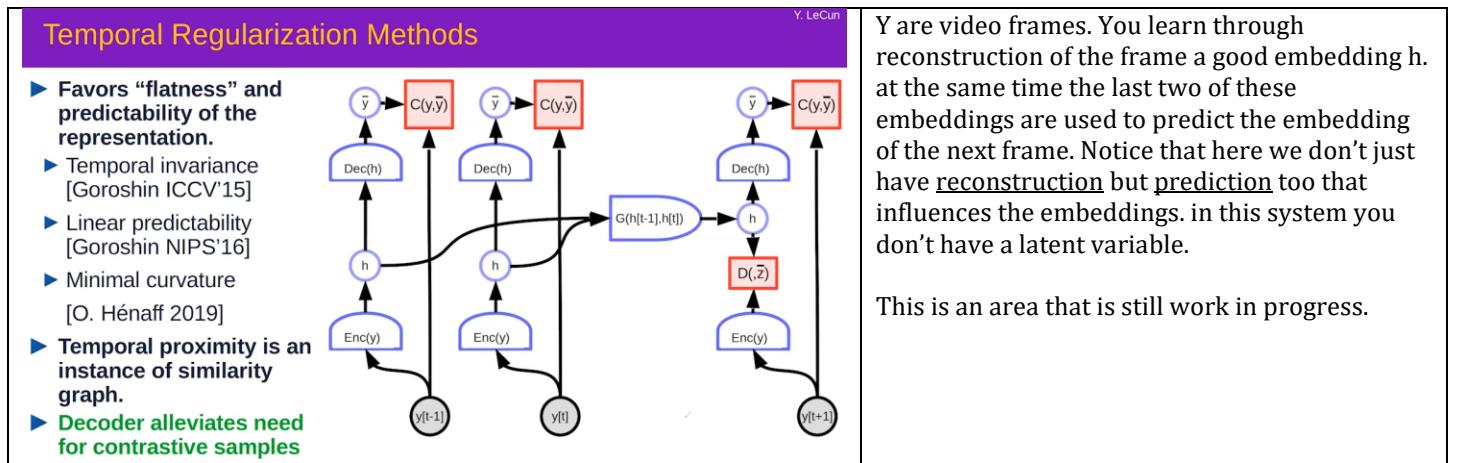
The prediction constraint is regularization! It means that whatever h you learn, it must be predictable. So it must be simple enough. For example, you might constrain the function G to keep half of h unchanged (as it was in the last frame, or the average of the last two) and be able to change the other half. This will force the system to embed all things that don't change in the first

half and all things that change in the second half. In the rotating image case, the content is the same, the rotation changes (the positions of the content). Essentially, you factorize h in an invariant and an equivariant part w.r.t the input.

The prediction function should be learned of course

It is another way for learning good representations (h) of visual features and use them in other CNNs for fine tuning.

Predicting video frames



Autoencoders

Autoencoders is a method for finding a good (regularized if you so desire) representation h , the embedding of the input. We no longer need to find z minimum with an optimization (minimize E to get the free energy) and then minimize the free energy. We directly minimize the free energy. Notice that the latent variable represents missing information while h is the complete representation of the input (although in the unconditional case with z , they look the same as I understand).

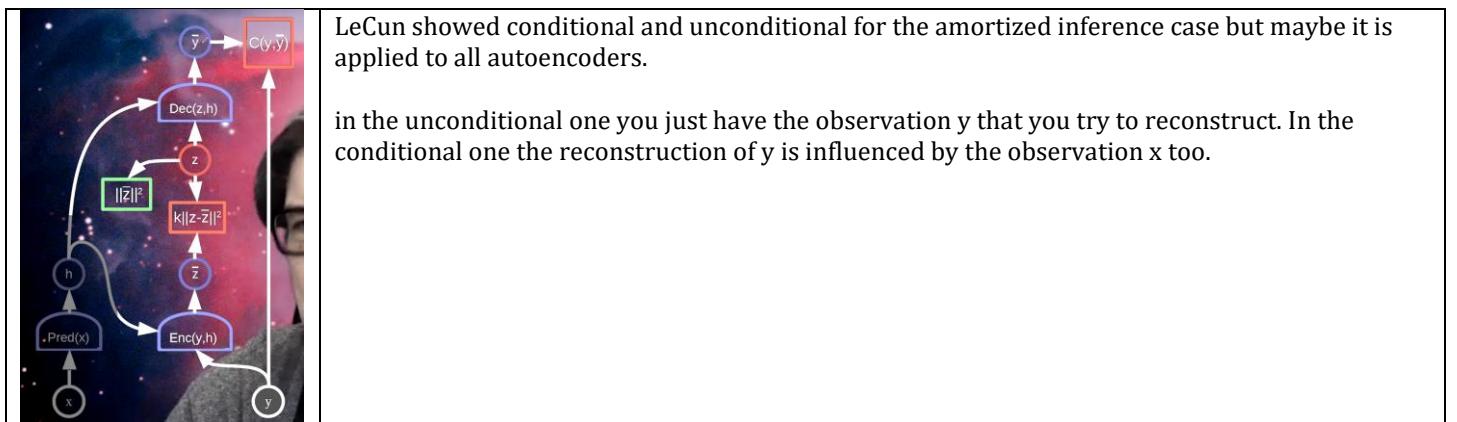
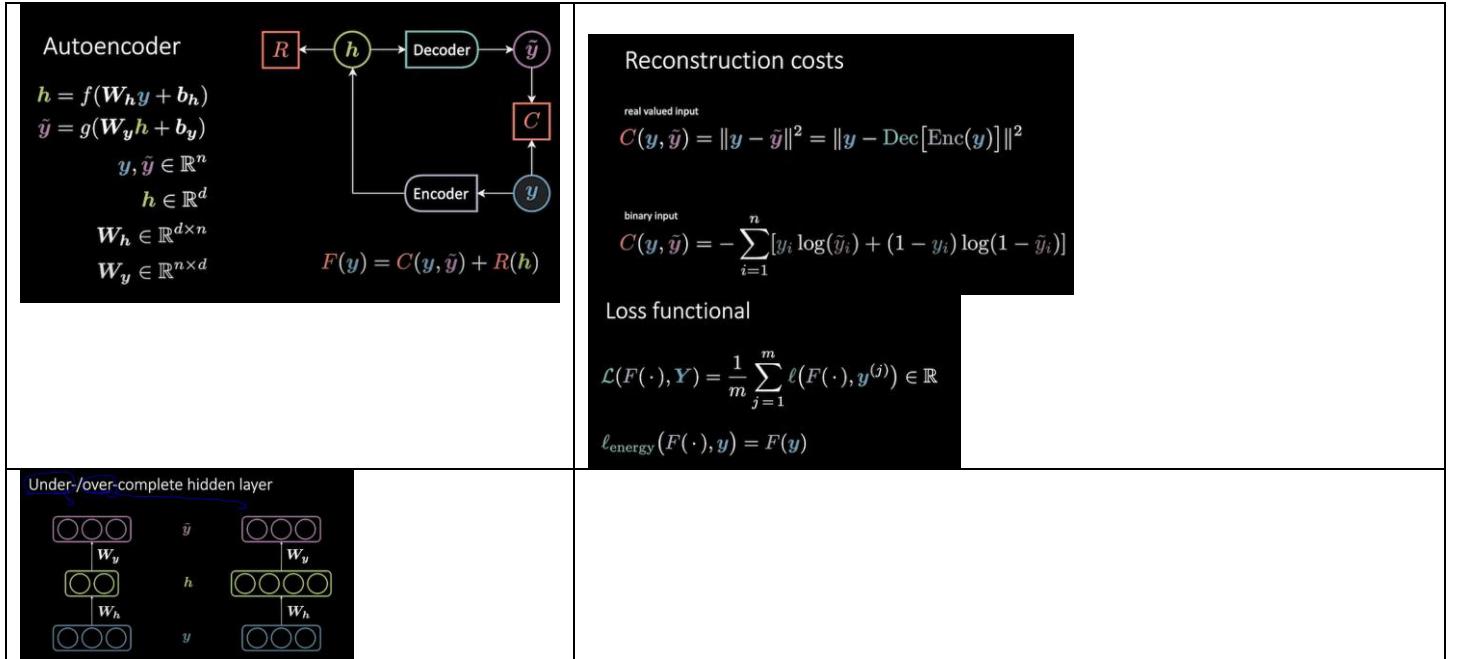
Autoencoders are better than GANs in image reconstruction, especially the discrete VAEs.

Notice that autoencoders aren't only used for creating compressed embeddings. You might want to extract an embedding that is higher in dimensions in relation to y (over complete hidden layer)! The reason is that higher dimensional data are easier to become linearly separable (easier for the decoder to learn as I understand). But in these cases you need to do some tricks so that the system doesn't simply learn the identity function (just copying input and making 0 the additional nodes) where the autoencoder can just reconstruct everything without learning anything though (energy collapse, 0 everywhere where it should only be on the training manifold). To avoid this problem when z is high dimensional, you have to make it sparse. A few ways to do this:

- Apply some regularization to h to make it sparse
- Add noise to y (Denoising autoencoders)
- Sample from a gaussian over y (Variational autoencoders)
- Contractive autoencoders

They are ways with which in reality you enforce the embedding (called code in AE) to have a low range of possible values it can take. You are making it sparse in other words. Only a certain amount of information can pass through it.

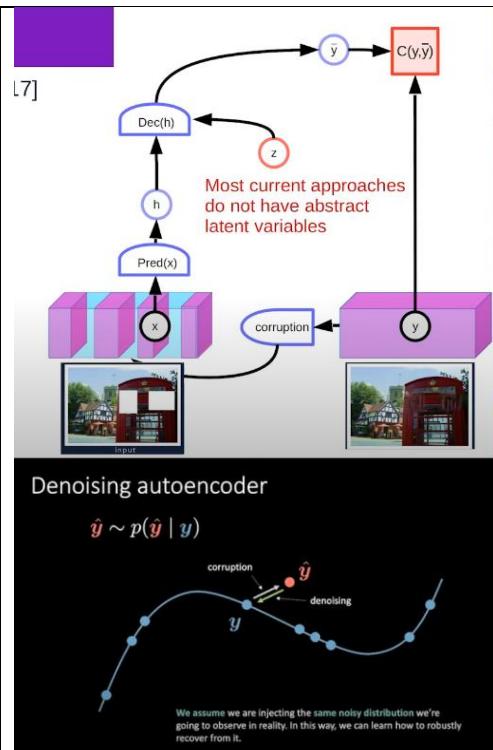
Typical autoencoder



Denoising Autoencoders

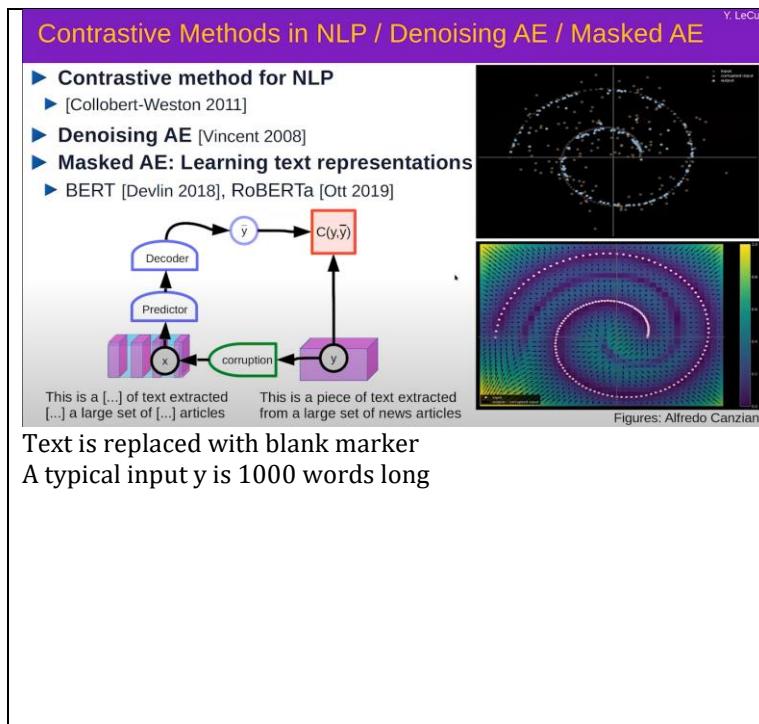
This technique of denoising AE works very well for pretraining natural language understanding systems. It doesn't work well for images. BERT is a famous DAE where the noise is a dropout, we drop some of the words (masked language model). In order to fill in the missing words, you need to understand the meaning of the text to some extent. It uses a transformer as the encoder (called predictor in the picture)

Notice that it is important that during inference the noise of the inputs should be similar with the noise the model was trained with.



This works very well for language because the way we deal with uncertainty of the output is to use a huge logsoftmax on all words of the dictionary, and the system outputs a range of words with scores for each one of them (that could represent probabilities if we want). For discrete data even for large cardinality like the while dictionary of a language you can use logsoftmax (a probability distribution essentially). But you can't do this if the output is continuous, for example the part of an image. In these cases you can't use logsoftmax. If you use the square error, then you will get the average of all possible outputs which would give you a blurred image patch. So despite the fact that denoising AE worked very well for language it doesn't work very well for images. For example if you apply it on image data, and you take the encoder's embeddings as input to an image classifier, it doesn't work well. The system hasn't learned good representations of the visual data. A solution to use denoising AEs for vision (and in all cases where you have high dimensional continuous output) is to use latent variables to represent uncertainty. Otherwise if you don't want latent variables you can use the joint embedding architectures which are very popular for vision (image recognition etc.). **Joint embedding architectures are better for vision because you don't have to reconstruct the image from the embedding.** You just have to learn the embeddings. So the system doesn't have to generate a high dimensional continuous output (an image). It just needs to learn proper low dimensional embeddings from images.

Contrastive learning



Denoising autoencoders can be framed as EBMs. (in the diagram the predictor network is the encoder that produces the representation of the input).

Here lecun describes how they are framed like EBMs trained with contrastive methods. The energy is the reconstruction error.

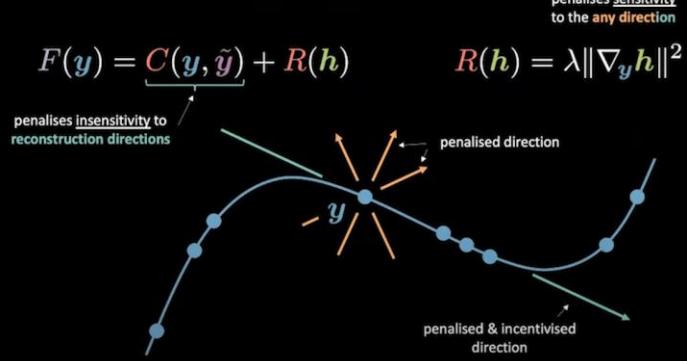
First you lower the energy of known points. By known I mean masked input where the input is known. The system produces the original one, and you give low energy to those pairs.

What about points to push their energy up? You pass in a masked y (not x, the original input is masked) and you don't noise it further. So $x = y$. the autoencoder will denoise x so the y_{hat} will be different than the noised original y. so you know that they will be different and you push these points up.

Me: If you define the loss as the energy, and thus the embeddings are learned in a way where there are areas that the embeddings should not go. You give them forbidden areas (those that produce ys with high energy).

Contractive autoencoder

Contractive autoencoder

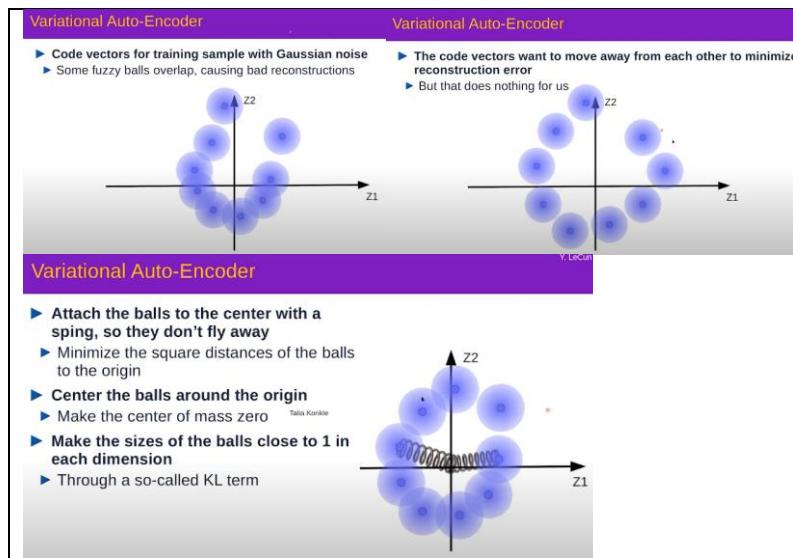


We add a regularization term to the cost. The cost becomes larger when the gradient of \mathbf{h} with respect to \mathbf{y} increases. This means that we want small sensitivity of \mathbf{h} to \mathbf{y} . If we toggle \mathbf{y} a lot, \mathbf{h} should toggle a little.

lambda is scalar, a hyperparameter.

Variational autoencoders (2013)

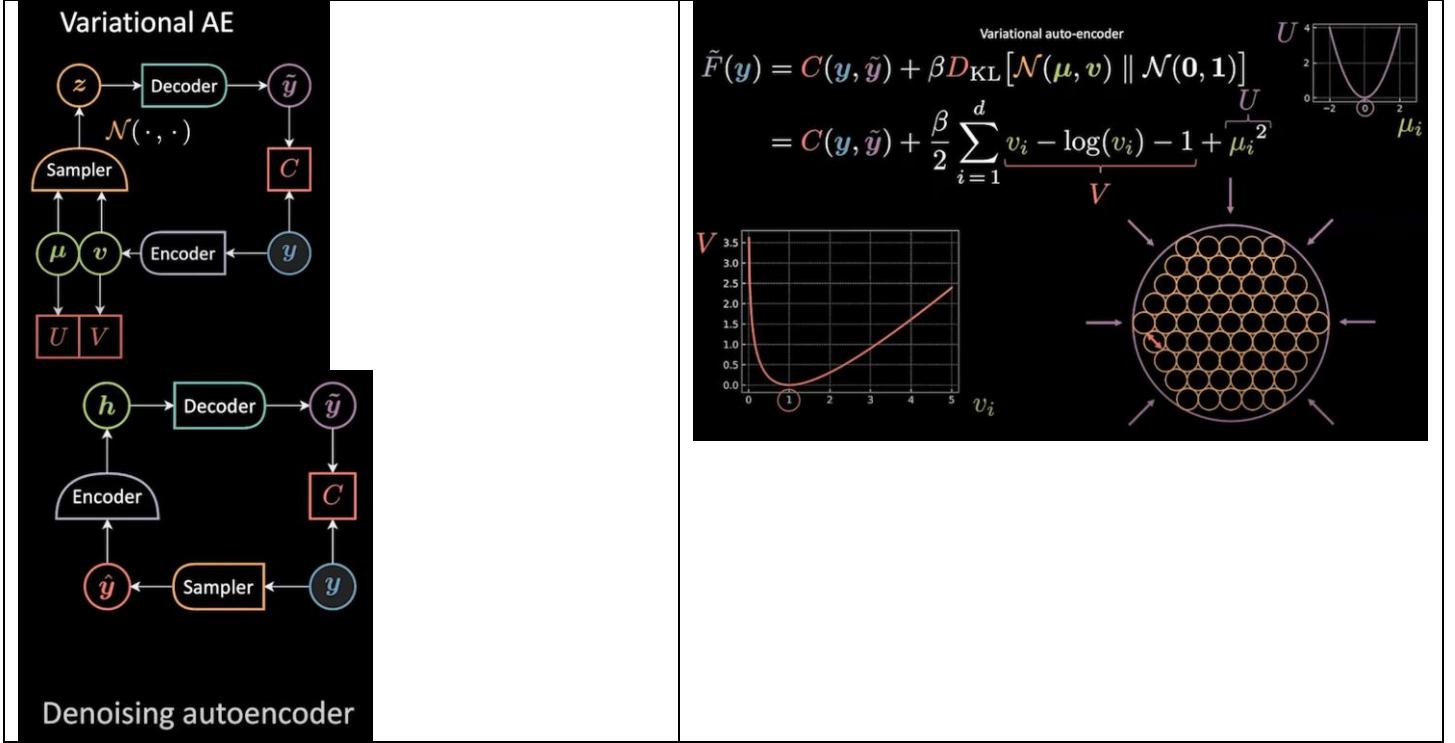
It limits the capacity of \mathbf{z} , because there are only so many bubbles that can fit within the big normal bubble.



The concept of variational AE is that instead of feeding the embedding \mathbf{z} directly into the decoder, you create a gaussian distribution over this \mathbf{z} and pick one \mathbf{z} from that distribution instead.

The problem though is that if the spheres overlap, the reconstruction will not be good. Thus the system will try to move these spheres away from each other.

We don't want them to go very far away so we kind of add a spring on them that holds them close to the origin. This spring is the regularization term.



The encoder generates two parameters μ and v of a gaussian distribution. the latent variable z is then sampled from that distribution. In the DAE we introduced noise in the input y while in the VAE noise in the hidden layer.

We want the distribution to be close to the normal ($m=0, v=1$) so that the latent variable is not powerful enough, we want it to be sparse. But we don't want it exactly normal. If all y s were converted to the normal distribution then no information would be retained. Everything would be the same. If the variance becomes 0 then you end up with a typical autoencoder since you only have the y points.

The cost function

If the bubbles overlap, then you don't know to which y a sample of the overlapping region belongs. The model might give y_1 but y would be y_2 . So C would be large. Notice that there are two ways to make the bubbles not overlapping. Moving them apart and reducing their variance (their radius). So we need something to constraint these moves. This is what the V and U terms do (V and U are regularization terms).

- The C term keeps the bubbles from overlapping.
- The V term keeps the bubbles from becoming 0 variance.
- The U term keeps the containing bubble as small as possible.

If the variance v goes to 0 V goes to infinity. If it goes to inf, V goes to inf too. V is minimized with variance 1. U is minimum when $\mu=0$ and goes to inf quadratically for increasing the μ . Notice that bubbles are not equal in size.

So we are searching for the minimum bubble with the maximum number of bubbles inside it. If we need to further increase the capacity of the latent variable we must increase the number of bubbles. This is only possible by increasing the dimensionality of z . for example we could make it 3d from 2d in this example. this would create a lot more space for additional bubbles. Notice though that we don't want a large latent...

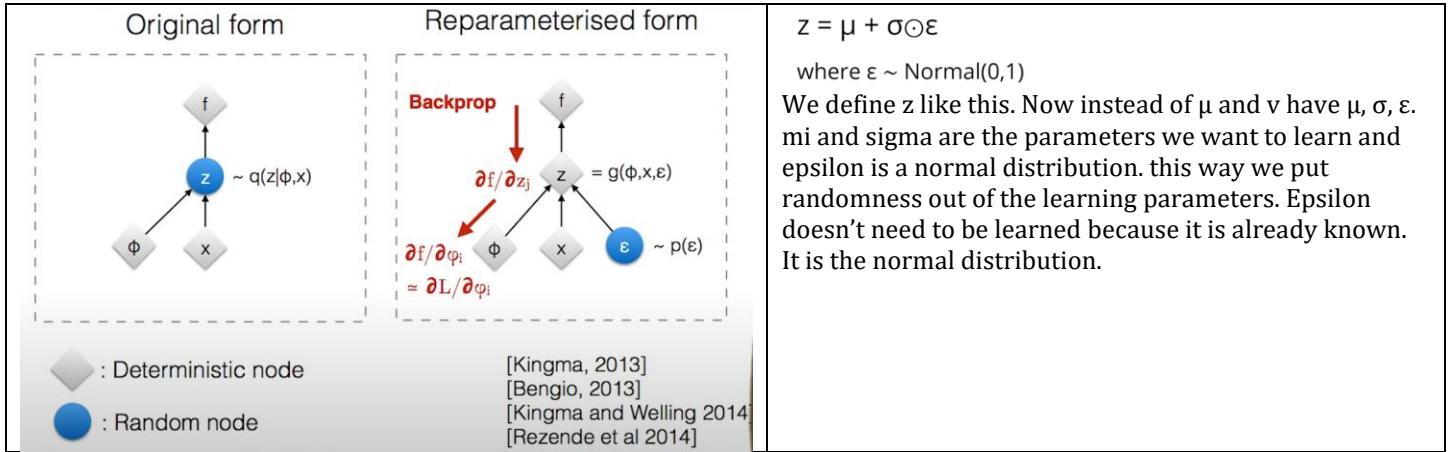
The KL divergence (D_{KL}) is a measure of the resemblance between two distributions (normal and given).

Disentangled VAE (β -VAE)

The β hyperparameter determines how much weight you want to put on the KL divergence. Somehow increasing beta makes the latent variable components disentangled with each other, meaning that each one of them controls a specific attribute of the output (x position, y position, zoom factor, etc.).

Reparameterization trick

You can't back propagate through sampling, so you re-parameterize the sampling with the variable z ... ????



VAEs are generative models

Notice that if you have a trained VAE, you can sample from the latent variable distribution (from the big bubble) and decode that sample. This will generate a new output (for example a new image) that doesn't exist in the training set. VAEs are generative models.

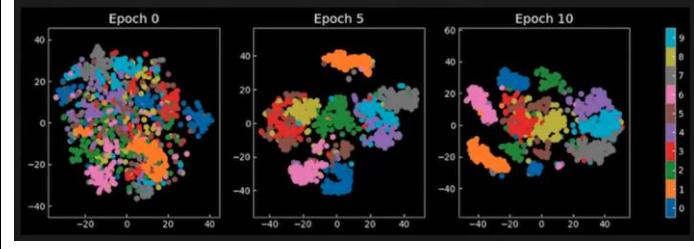
Linear interpolation of two different embeddings and then generating a sample can create a chimaera.

MNIST example

		<p>The area outside of the digit, has low spatial frequency meaning it is a uniform region of pixels (with value 0 or very close to it in this case). A typical autoencoder will learn to generate salt and pepper noise for these areas (second picture) so that their sum is cancelled out.</p>
		<p>A DAE on the other hand, learns to ignore what happens in these low frequency regions. The reason is the noise added to the input. now these regions are not of low frequency they have some random patterns that don't contribute to the cost though. So it leans to ignore them completely adding 0 weights in these areas.</p>
		<p>This is a new output generated by sampling z, on a VAE trained on MNIST. Why it generates something that is not a digit? A possible reason is there is a void in a region of the big normal bubble. As I understand it, in training the VAE learns proper latent variable values that can generate real digits as they exist in the training set.</p>

Theoretically if the bubble was full, then any sample would generate plausible digits. If you have generated outputs that are really bad, then this might suggest that there is a void in the bubble, a region that was not covered during training and so it doesn't produce valid outputs. In training, the rest of the bubble was filled and all samples were reconstructed from that region of the bubble. (So in order to validate a VAE you need to sample it and check the outputs?)

The reason that the big bubble has a void, is probably because the reconstruction cost C is dominant, so that U can't push adequately the small bubbles close to 0 and V can't enlarge them adequately. If the model has a lot of outputs like this then the void is significant, and we might have to fine tune C , V and U (as hyperparameters) and retrain. So maybe plot all three components of the cost separately to see what's going on.



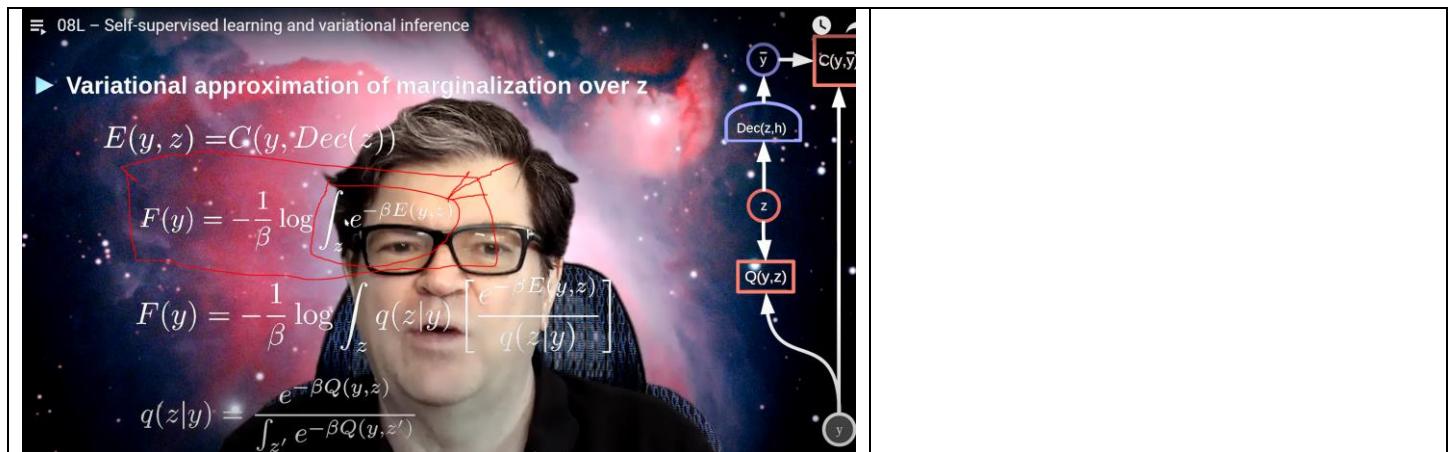
This is an actual bubble with bubbles. The z in this case is 10 dimensional. Here we plot z in a 2d projection. As the model trains, the embeddings of the same digit are coming together. The model learns to represent this digit with a certain embedding that lies in that region. Notice that the embedding clusters are not touching each other in 10 dimensions (and even in their 2d projection some of them don't touch each other) due to the reconstruction error term that tries to create non overlapping bubbles.

To debug in ML you should plot a lot. Some plots for VAE:

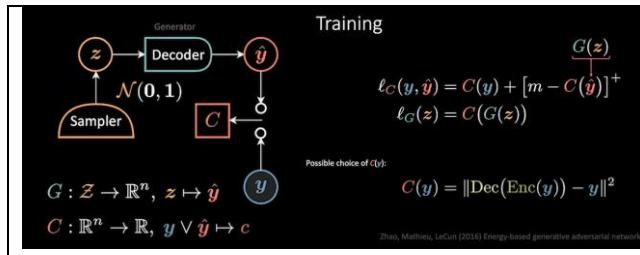
- Some outputs per epoch during training
- Generating sampled points
- Interpolate between embeddings and plot the interpolation
- Z clusters per epoch

Math of VAE

Just have in mind that VAEs are a result of some complicated math. The free energy contains an integral that is intractable (can't be computed efficiently) so we replace it with a variational approximation of it based on a few mathematical tricks. This variational approximation leads to a free energy that has also an entropy term which varies with β (actually with temperature). So you can not only use the zero temperature limit free energy where only one z (the z-check) contributes to the free energy, but you can use more relaxed free energy by selecting a range of possible zs that give low energy and the free energy will be the average of them.



GANs



You sample from a normal distribution. The sample goes to a generator which creates a y-tilde. The y and y-tilde don't get together into the cost module (this is what this diagram shows). First y-tilde gets in, then y, one after the other. This is the system. The goal is of course to minimize the loss function. The cost function is a sum of 2 terms. $C(\mathbf{y})$ and $m - C(\hat{\mathbf{y}})$. We want the first to be small (the "real" y gets a small cost C). We want $C(\hat{\mathbf{y}})$ to be high optimally as high as m (the "fake" y-tilde gets a large cost C).

But the generator is trained by this cost function C (not the whole loss function). So it tries to generate y-tilde that make C small.

This means you have a battle between the generator and C. And the problem is that when you have 2 conflicting optimization problems like this, the best solution (best in total) is not stable. It is a Nash equilibrium, which means that if you move a little bit off of it, the total will become larger (smaller for one, but more larger for the other so the sum will be larger). There are various tricks proposed to tackle this problem.

C can be a network on its own. It can be an autoencoder. In this case it tries to reconstruct nicely the real y and don't reconstruct nicely the fake y (it tries to learn to distinguish them). This is the critic network.

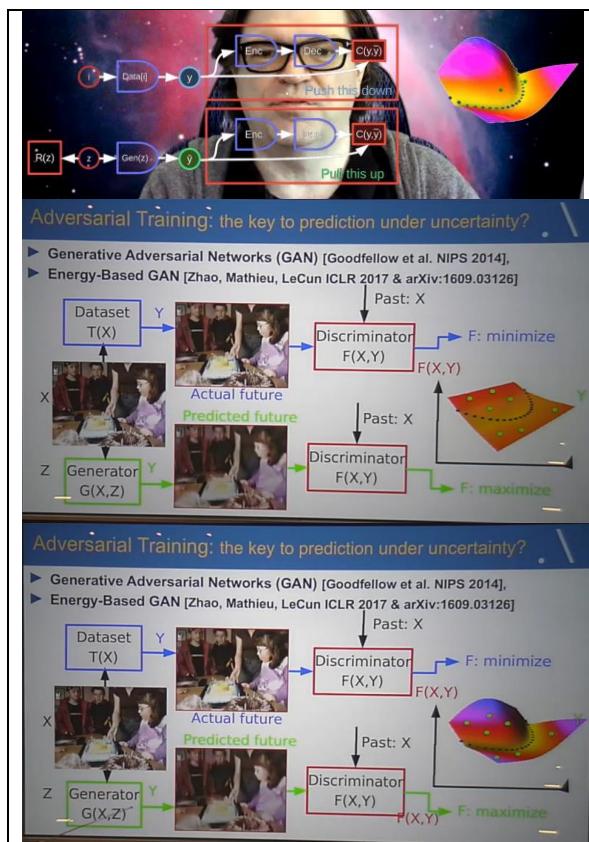
If you use a discriminator instead, it is a classifier. But it is better to use a critic.

A GAN is actually a contrastive method for training EBMs

<https://www.youtube.com/watch?v=8u2s64ZtmiA&list=PLIHTzKZzVU9e6xUfG10TkTWApkSzCzuBI&index=14&t=1756s>

You train the generator to produce points that are close to the data points (but they are not the actual datapoints) and you push their energy up. The critic (or discriminator as it was used to be called) is now the energy model (the red box). It says if the sample is good or not. But you don't want to have a very high slope in the energy function. This is what will happen eventually. So you have to regularize the slope so that the surface is smooth. This is what Wasserstein GANs (2017) do.

Notice: Till 2022 at least, there is no success in using GANs to pretrain systems in a self-supervised manner (as you do with joint embedding for example, see Wav2Vec). They are only good for generating content like images or sounds.



Energy based GANs, adversarial training

The input is a video frame. You train the discriminator to produce low values for the output vectors that it observes.

Then you take the generator using a randomly sampled latent variable z from a certain distribution to make a prediction of what is going to happen. Initially it would be wrong. You use the knowledge that they are wrong to train the discriminator to produce high values for these output vectors. You do this with a proper loss function that gets smaller when this cost gets larger, for example $\text{loss} = C(\text{good}) + (m - C(\text{bad}))^+$. So after a while the discriminator learns the difference between what is wrong and what is plausible. It learns the shape of the energy function.

Then, after we have constructed the energy function, we have to train the generator to produce plausible predictions. Using backpropagation you can figure out the gradient of the discriminator output (the gradient of the energy function) with respect to the parameters of the generator. So the generator can be trained to produce outputs that minimizes the discriminator's output which means that the discriminator identifies them as plausible.

Mode collapse

This is a big problem with GANs. The critic doesn't give meaningful gradient to the generator and the gen. keeps creating the same. Essentially the energy function becomes too steep (very low around the datapoints and very high right next to them). If you train a GAN for long enough this is what will happen. So you have to stop before this happens.

The reason is that you are minimizing two functions at the same time (loss w.r.t the critic and w.r.t the generator). So you are actually searching for a saddle point (or a nash equilibrium). Do this for long enough and one of them will win and you have mode collapse.

Structured prediction

Structured prediction is when the inferred output has some structure, for example it is a word from a dictionary, a sound from a language etc. it is subjected to certain constraints. In this case it is not enough to just recognize the characters and output a sequence of characters. This sequence must be a word from a language dictionary. This can be applied to speech recognition as well.

So when you have these two needs, **you must use a latent variable predictive model.**

1. Multiple plausible outputs for a given input
2. And the output has some structure in it

Examples

you have strong dependency between the variables you are trying to predict

- A description of an image, it must satisfy grammar etc.
- A list of objects that are in the image (for example DETR model)
- Translation of a sentence (there are multiple plausible translations) it must satisfy grammar etc.
- Image denoising
- Video prediction (of the next frame)
- Biological sequence analysis
- Speech recognition, handwriting recognition (the same)

Handwriting recognition example

➤ Intro

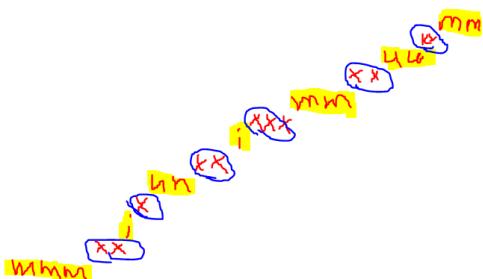
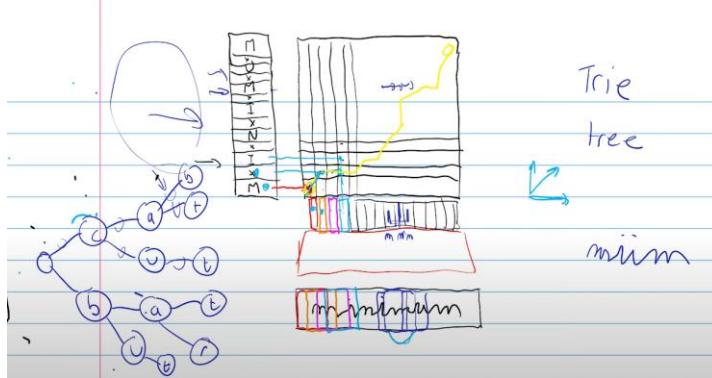
When inference involves latent variables	<p>➤ Latent variables are variables whose value is never given to us.</p> <p>➤ Examples: to read a handwritten word, it helps to know where the characters are</p>  <p>➤ To recognize speech, it helps to know where the words and phonemes are</p> <ul style="list-style-type: none"> ▶ You can't read this if you don't understand english ▶ Vous ne pouvez pas lire ceci si vous ne parlez pas français 	<p>Z controls the boundaries between characters. X is the input image. Y_hat is the set of scores for the bounded regions of the sequence (to which class they belong). So y is a proposed set of classes (a proposed set of characters, or a proposed word).</p> <p>I guess that the energy function can have been constructed with a language model, so it gives low energy to words that are observable (words of the dictionary). This is where the structure of the prediction comes into play.</p> <p>The language model is contained in the decoder of the diagram.</p>
---	---	--

It isn't enough for the system to recognize a sequence. It must be a word of the dictionary in order to have a low energy.

The system finds a combination of a set of classes and a set of boundaries that for a given image x, it gives low energy (an output that satisfies some constraints)

This can be applied to speech recognition as well. The bounded regions must be words of that language

➤ Training



An example of a valid prediction. 3 m in a row mean that the first 3 windows recognized the m so we are very certain that the first character is m. if the predicted word doesn't match the label, we back propagate gradients.

In training you know the label (the correct characters), but you don't know the location of each character, where they are separated with each other. So we will create a latent variable that controls the separation between characters, and we will find the correct latent variable value so that output classes match the label classes. (a z value produces a set of boundaries. Another value produces a different set of boundaries. One of them will be the correct one).

The concept is similar to DETR model, where we have the list of objects in the image as labels, but we don't know where they are and we use a latent variable for it.

The label is the correct classes for the characters, here minimum separated by the blank character. The blank character is used as an extra class apart from the 26 classes of letters so that the system outputs this blank character if it is unsure. The blank between two characters in the label assumes that the system will not recognize a character when it looks at a window on the boundary between two characters.

There is a CNN that looks the whole input image. It has many windows. Each window looks at a small part of the image and outputs a class for it. the window has some context, it is wide enough to get some context.

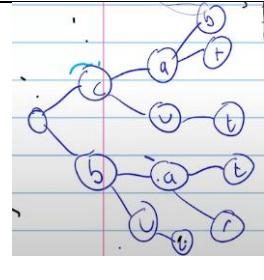
Each position of the grid shows the distance between the predicted class and the label class. This distance is the cost. Starting from lower left to upper right we select points with the smallest cost (the smallest distance). We can move right, up or diagonally. We select the first rectangle. It has a high score. We move up, right or diagonally to one of these three positions that has the lowest cost. And repeat. We end up with a predicted word. The goal is to minimize the sum of the cost along that path. For example if all characters are correct except the first one "ninimum" the cost of that path will be higher than the prediction "minimum" because the first position m-m has lowest cost than m-n.

➤ Inference

In inference you don't have a label which means that you try to find both the shortest path and a sequence of letters than minimizes the cost (that produces the shortest path, the path with the smallest cost). You must consider that valid sequences are only the words of the dictionary, not random letter sequences. One way to do it would be an exhaustive search where you check all words and get the shortest path (a cost) for each word and then you select the word with the smallest cost (the shortest shortest path). But we don't do it this way of course because it is inefficient (English has 200k words).

Another way is to represent the dictionary not as a set of words, but as a tree. Each node is a letter. And each letter is followed by the letters that usually follow the previous letter. For example in English q is usually followed by u, so the cost from q to u will be small while the cost from q to l will be large. The edges give the cost (how common it is that this word follows the previous). This way you can represent the whole dictionary. It is an alternative way of representing a language dictionary. then you must find a good shortest path together with one of the branches of this tree.

Notice: This tree representation is possible because the words in a dictionary are a discrete variable. So inference is efficient. You do it by finding the shortest path on a graph (Viterbi algorithm) which has linear time complexity $O(V+E)$. it is not an exhaustive search. (The simplest form of grammar, dependency between the variables you are trying to predict, is that a word y_1 could be followed by a subset of all words.)



Suppose that this is your dictionary. it only has 6 words (cab, cat, cut, bat, bar, but). You have a trained model that can read handwriting from an image. You give it an image. You don't have a label (the target classes). You begin from the first location of the "grid" (from the first prediction) and compare it with the possible first letters of the dictionary which are either c or b. You pick the one with the smallest difference from your predicted class (you pick the smallest cost). You repeat this process but now with the second possible letter which is a subset of the whole dictionary. Notice that you have a latent variable. So you minimize cost w.r.t it too, which means that in inference you try to find the best segmentation for the letters, that match an existing word of the dictionary.

In general, this is the way to do simultaneous segmentation and classification.

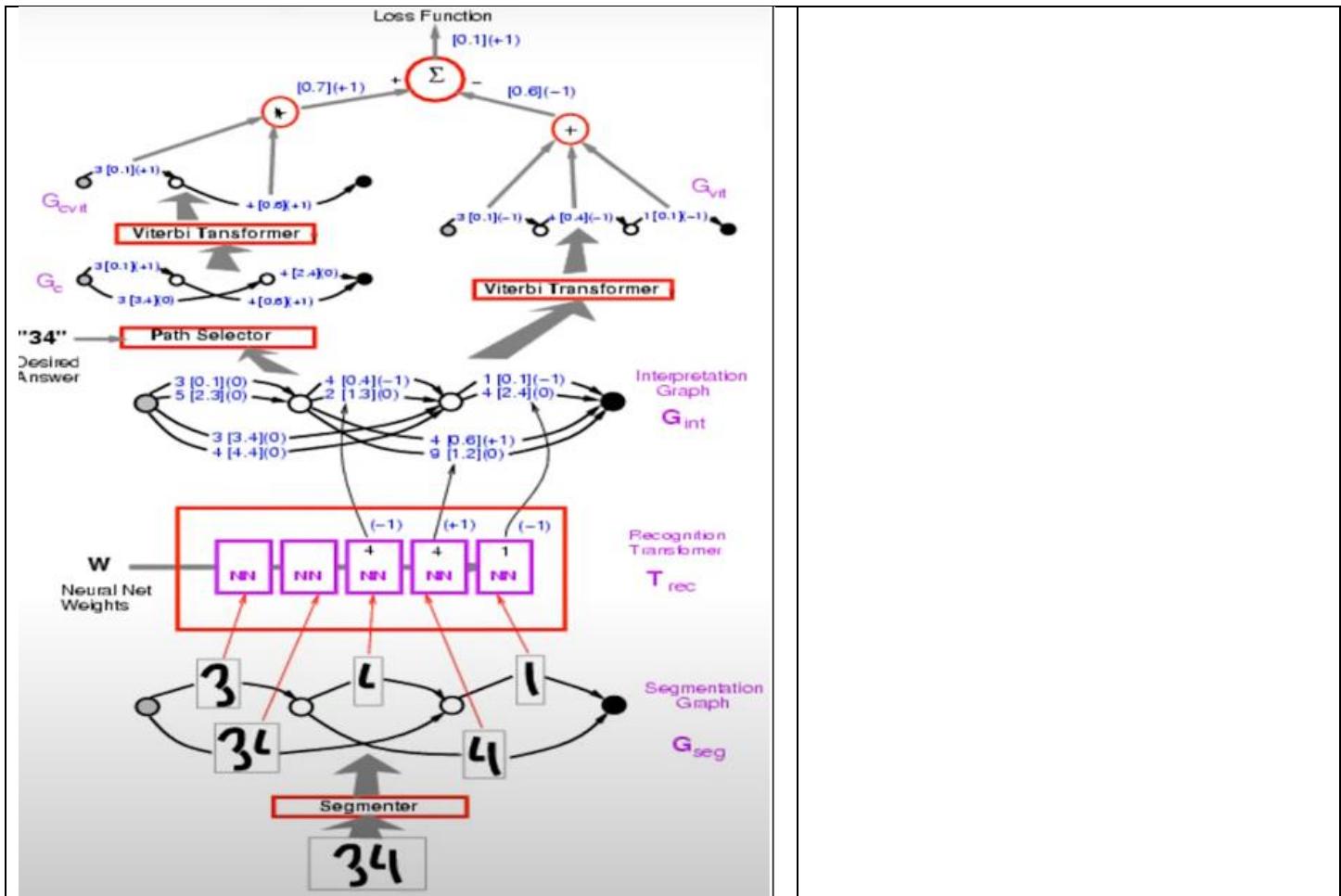
Notice that the graph is only used for inference.

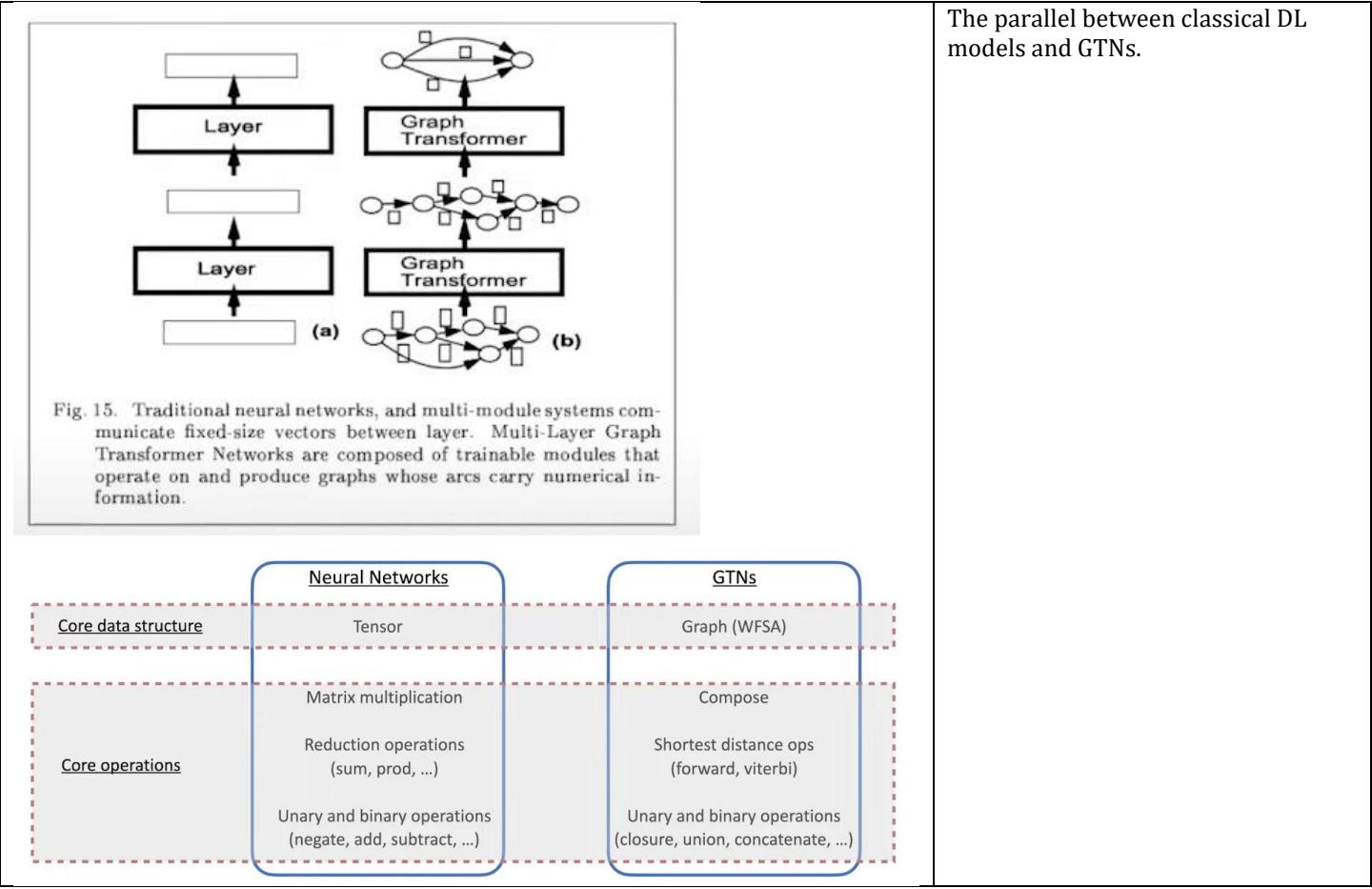
GTNs

<https://github.com/gtn-org/gtn>

As in the handwriting recognition example, where the dictionary can be represented by a graph of letters, there are cases where the hypothesis from a recognition system can be best represented by a graph. It's a Mid 90s idea and it is called Graph Transformer networks, GTNs. They are not transformers, nor graph networks. The state of the system is not represented by tensors or vectors, but by graphs where the nodes and transitions are attached values to the graph. GTNs are implemented in pytorch in the GTN library. It can backpropagate gradients with respect to the cost, through path selectors.

Graph Transformer Networks	
<ul style="list-style-type: none"> ▶ Example: Perceptron loss ▶ Loss = Energy of desired answer - Energy of best answer. ▶ (no margin) 	<p>This is a similar handwriting recognition example (mid 90s) where we do simultaneous segmentation and classification (although without using a moving window in the CNN).</p> <p>We have two phases in order to construct the loss. First is the clamped case in which we give the right answer and we select the path for that.</p> <p>Then there is the unclamped phase where we just do inference on the same input. we get a different prediction (the one with the smallest energy).</p> <p>We want to increase the energy of this wrong prediction and decrease the energy of the right one. We pick their difference and set that as a loss function.</p> <p>(This example is a typical latent variable EBM, where the latent variable controls the segmentation of the image)</p>





Why to use GTNs

Why Differentiable WFSAs?	WFSA Weighted Finite State Automata
<ul style="list-style-type: none"> Encode Priors: Conveniently encode prior knowledge into a WFST End-to-end: Use at training time avoids issues such as label bias, exposure bias Facilitate Research: Separate data (graph) from code (operations on graphs)! 	

GCNs

<https://arxiv.org/abs/1609.02907> (Original paper 2016)

https://www.graphneuralnets.com/p/introduction-to-gnns?coupon_code=GAT the best one on graphs

<https://www.youtube.com/watch?v=Iiv9R6BjxHM>

<https://www.youtube.com/watch?v=M60huxIvKbE>

https://www.youtube.com/watch?v=tuChBS08_eg

<https://web.stanford.edu/class/cs224w/> Machine Learning with Graphs

https://www.youtube.com/watch?v=JAB_plj2rbA&list=PLoROMvody4rPLKxIpqhjhPgdQy7imNkDn CS224W Machine Learning with Graphs in youtube full course

<https://www.dgl.ai/> DEEP GRAPH LIBRARY Easy Deep Learning on Graphs

Why Graphs

Suppose we want to predict if a new account is a fraudulent account or not based on the data it provides to us during registration, like its IP address and its credit card. (Or if a new tweet is fraudulent or not). You could use a typical DL approach, where one feature is the credit card number and another the IP. Although, this would probably not work because there are too many credit card numbers where each value is used just a few times or even once. Think of it like this: (me) There is no structure in the credit card number itself that would make a card fraud or not. A more appropriate feature would be the number of fraud and non-fraud accounts that have used that credit card in the past. Or we might add IP information of those accounts. We can engineer features like this, and train a typical model, but the point is that these relationships are by default encoded in a graph structure and we can use these by default. Using a graph, we can use additional information from other relationships that might be important but not considered if manually engineer a feature.

With graphs we can use a learning algorithm in order to exploit the fact that the data has local patterns which are important. The local connectivity of a node is very important for predicting its "value". Its behavior is influenced by its neighbors. In typical images neighbors share some spatial location, in graphs neighbors share some relationships (the edge of the graph represents some type of relationship, follows, blocks, owns, etc.). For example nodes that follow the same node are neighbors.

Some powers

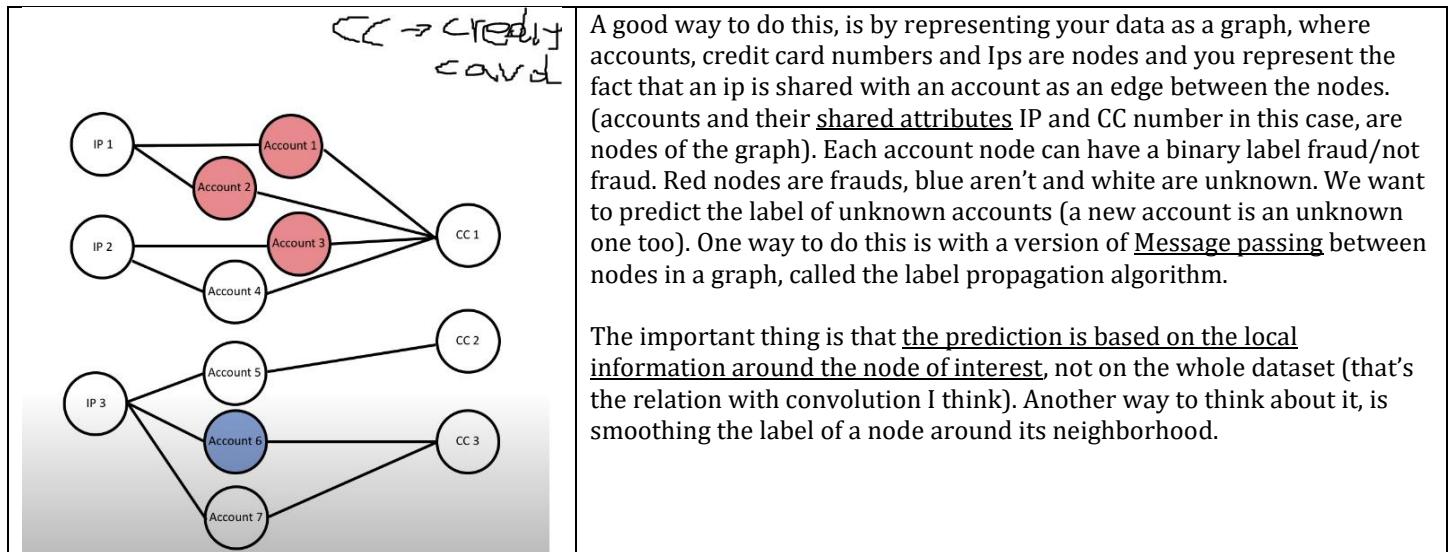
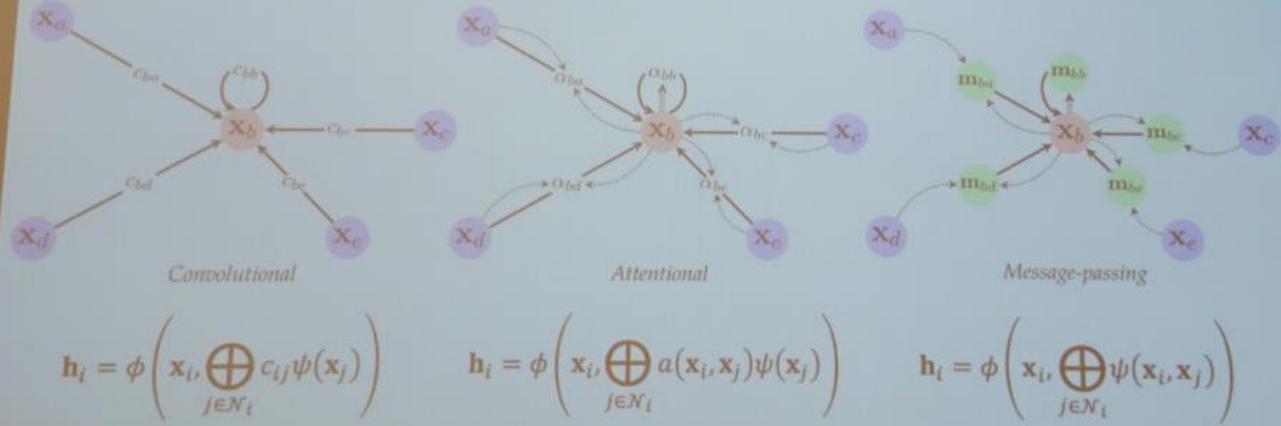
1. They encode what interacts with what
2. They are permutation invariant (for the same neighborhood you have the same output regardless of how the nodes are ordered within the neighborhood)
3. Data efficient. They have a lot of examples (the node neighborhoods) that all share the same network parameters. You don't only have one label for all nodes like in a picture classification example. you have labels for all the nodes of the graph and this large number of errors can lead to efficient learning.
4. It can generalize irrespectively of the graph size. Because the behavior of node is determined by the neighborhood so if it learns that interaction well, it can apply it to large datasets. Properties of the physical system are encoded into the learning problem itself. This is called inductive priors.
5. Interactions can be discontinuous in a dynamic graph. For example, in the particles simulation, the size of the neighborhood of a node changes as the process evolves and the article moves.

Easier to parallelize due to locality. Gpus have less data to exchange in relation to a random graph. Also, the model is applied to every node independently. For parallelizing a CNN you have to use other techniques like pipelining or model parallelism which are harder to wrap your head around.

Some limitations

- If you have long range forces a GCN can't capture them easily
- If you have 3 way interactions instead of 2 way interactions between the nodes (not sure what he means) they a GCN can't handle that easily

The three "flavours" of GNN layers



Convolution on a graph

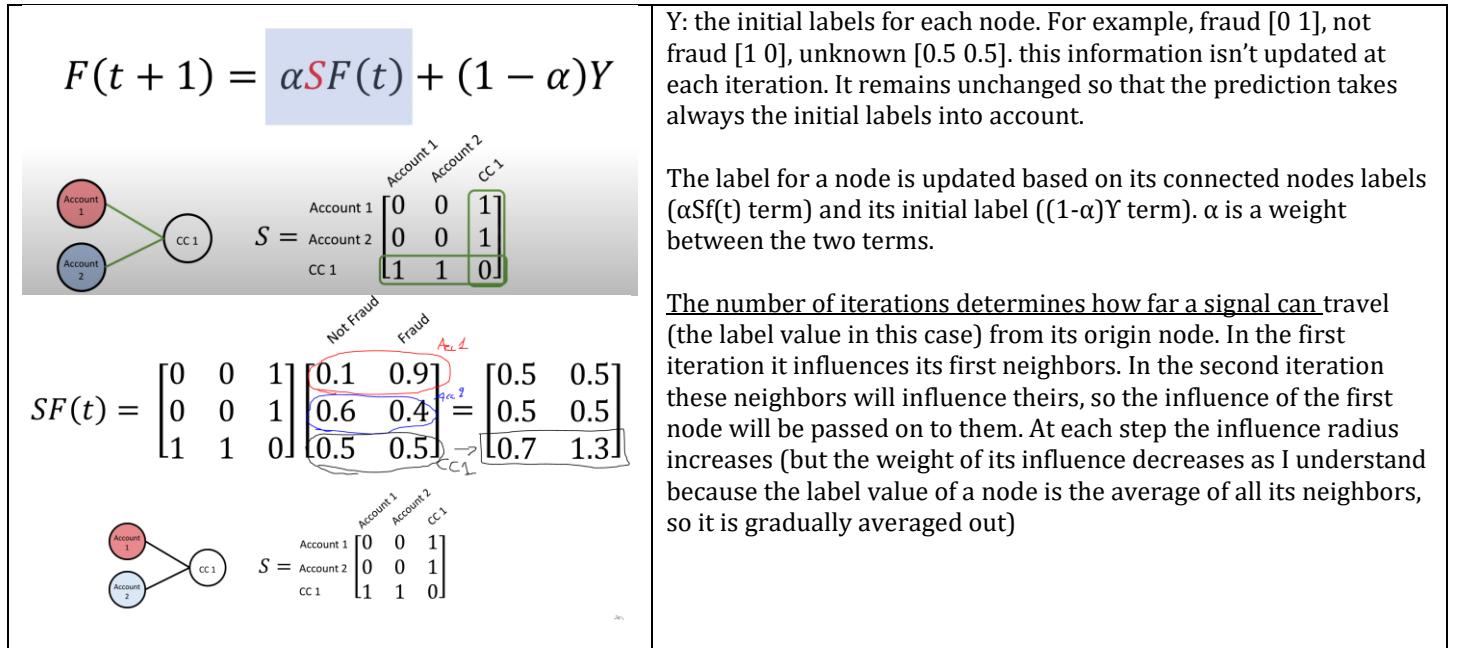
To do

Label propagation

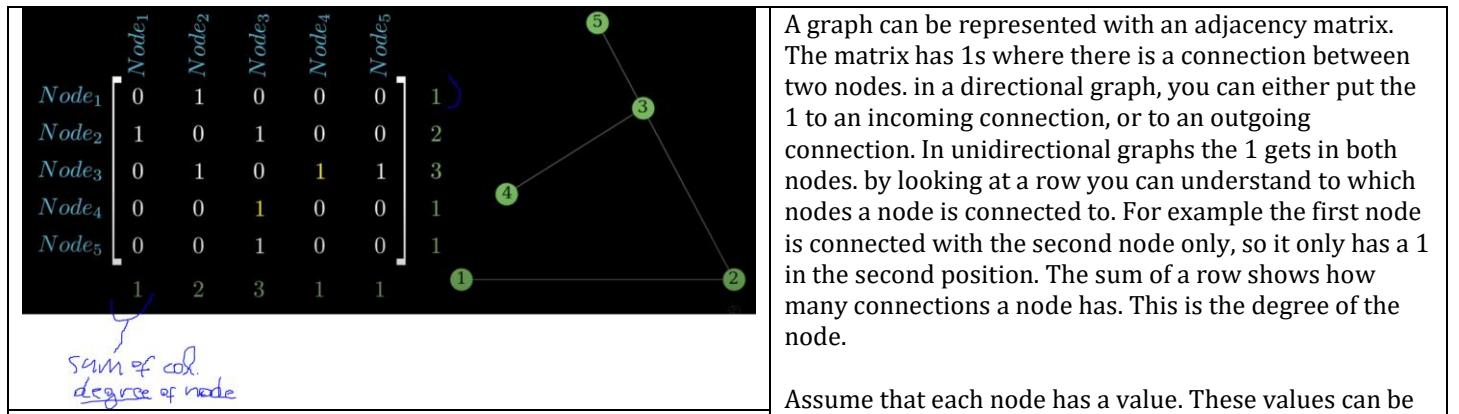
We have a label that describes some of the nodes in the graph. The label is a binary value, "fraud" and "not fraud". We want to predict the label for the unlabeled nodes, based on their relationship with the rest of the nodes.

This is called label propagation and it is based on a matrix multiplication. Essentially a node's label is the average of the labels of its neighboring nodes (this is what the multiplication of S with F does) and we add to that the initial label. Both terms are weighted.

$F(t)$ is the labels matrix at state t . $F(t+1)$ on the next state. (Notice that this operation can be applied to the whole graph so all nodes are being updated, the known too as I understood, but taking the initial labels into account). S gets value 1 if there is a connection between the corresponding nodes. In this case we have an undirected connection so S is symmetrical.



Adjacency matrix



Message passing

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix} = \begin{bmatrix} (0)(1) + (1)(2) + (0)(3) + (0)(4) + (0)(5) \\ (1)(1) + (0)(2) + (1)(3) + (0)(4) + (0)(5) \\ (0)(1) + (1)(2) + (0)(3) + (1)(4) + (1)(5) \\ (0)(1) + (0)(2) + (1)(3) + (0)(4) + (0)(5) \\ (0)(1) + (0)(2) + (1)(3) + (0)(4) + (0)(5) \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 11 \\ 3 \\ 3 \end{bmatrix}$$

Assume that each node has a value. These values can be represented by a vector. The vector [1 2 3 4 5] indicates the values for each node of the graph. We can apply message passing, by multiplying that vector with the adjacency matrix. The result shows the new values for each node. This is message passing in graphs ($A*v$), where the nodes pass their value as message to their neighbors.

By multiplying a vector with the adjacency matrix, we aggregate the neighborhood values for all nodes of the graph, at once. If we scale that matrix by $1/\text{neighbor-size}$

$$D = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad D^{-1} = \begin{bmatrix} 1.0 & 0 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0 & 0 \\ 0 & 0 & 0.3 & 0 & 0 \\ 0 & 0 & 0 & 1.0 & 0 \\ 0 & 0 & 0 & 0 & 1.0 \end{bmatrix}$$

$$D^{-1}A = A_{avg}$$

the aggregation becomes averaging, if not it is a summation. to average we use the degree matrix which is a diagonal matrix with the degree of each node. We invert it so a node with degree 3 has value 1/3. We multiply by the A matrix and we get a new adjacency matrix A_{avg} the aggregation of which is averaging.

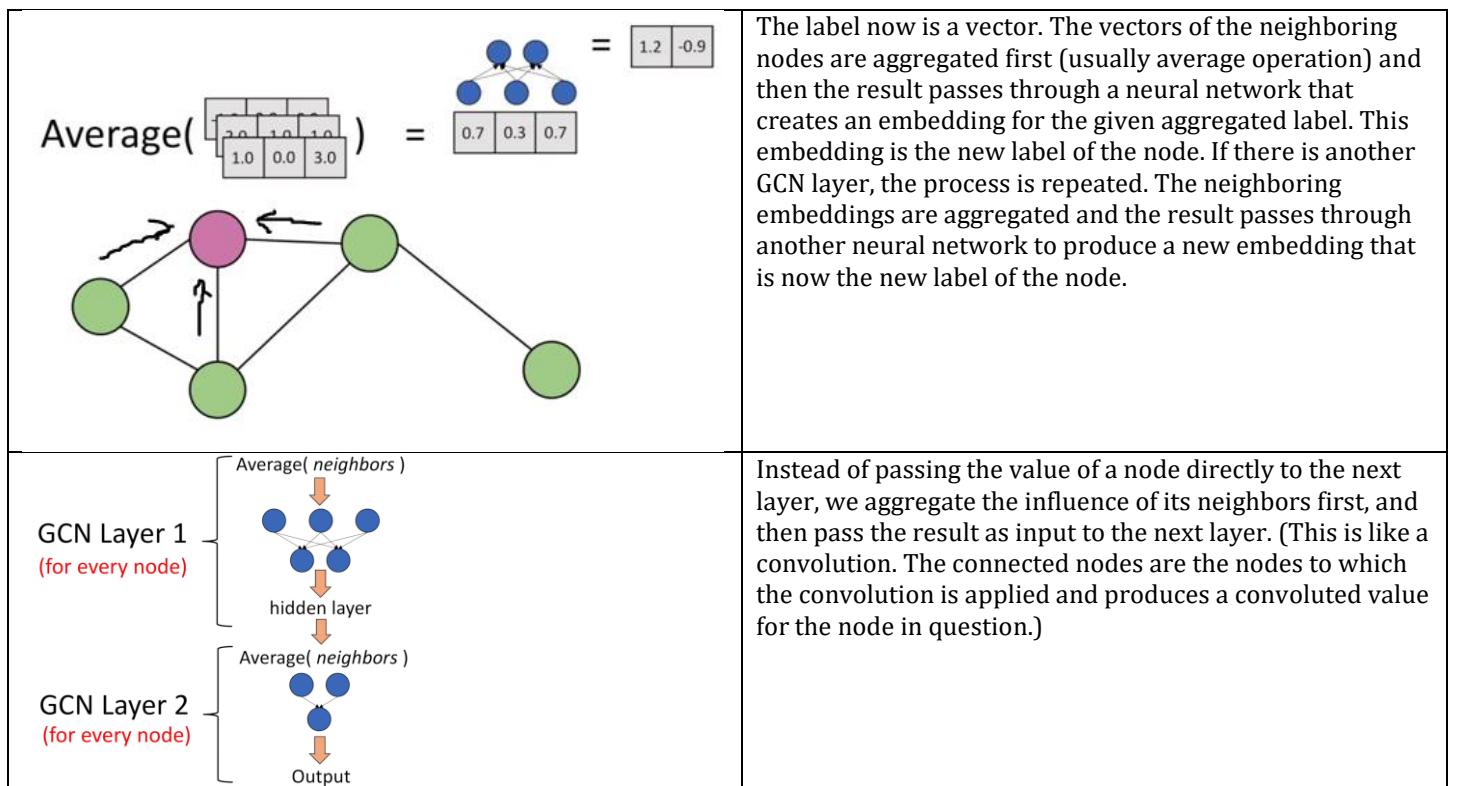
$$\begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.5 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.3 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0.0 & 1.0 & 0.0 & 0.0 & 0.0 \\ 0.5 & 0.0 & 0.5 & 0.0 & 0.0 \\ 0.0 & 0.3 & 0.0 & 0.3 & 0.3 \\ 0.0 & 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 & 0.0 \end{bmatrix}$$

In GCNs you include the nodes own value in the aggregation. This is reflected in the graph as a self-connection of the nodes.

Scaling properly the values of the adjacency matrix (for example averaging over the number of connections) is important to keep numerical stability when repeatedly applying that matrix. We want a matrix that provides numerical stability when applied multiple times, instead of creating exploding numbers or decaying to zero.

By repeatedly applying the adjacency matrix, the signal repeatedly moves to more distant neighboring nodes. We can repeat until we have a steady state where further iterations will change the value for each node very little, less than a threshold as I understood (why there is a convergence?)

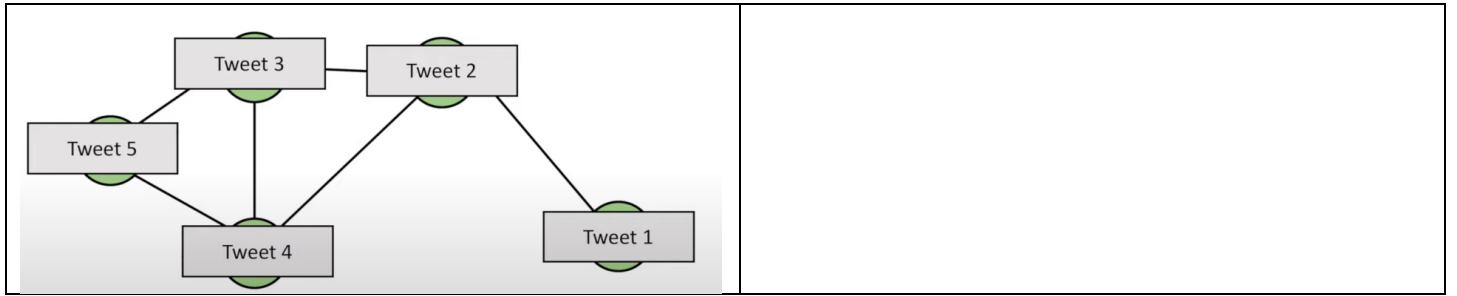
GCNs (Graph Convolutional Networks)



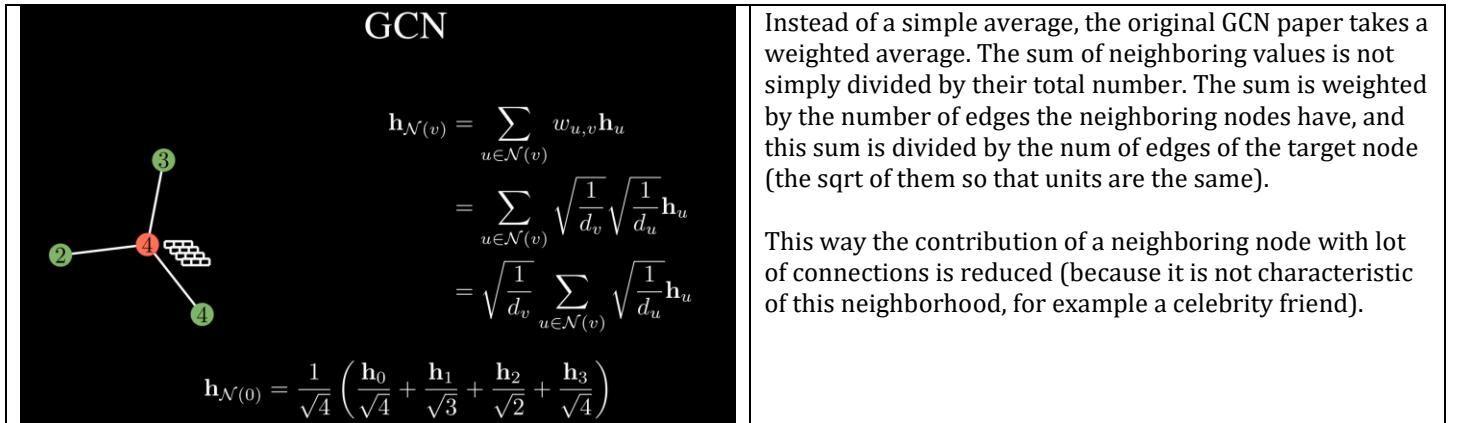
Notice that the neural net is the same for all nodes. Its not that each node learns its own network. There is only one W matrix (for each layer). Each layer of the GCN has the same weights for all nodes. It is the same layer, the same network. The

weights are shared. The input to that common layer is local. It is as if it is a CNN kernel. A kernel has some weights and it looks at small regions of the input. but it slides to the cover the whole input. something similar happens in GCN.

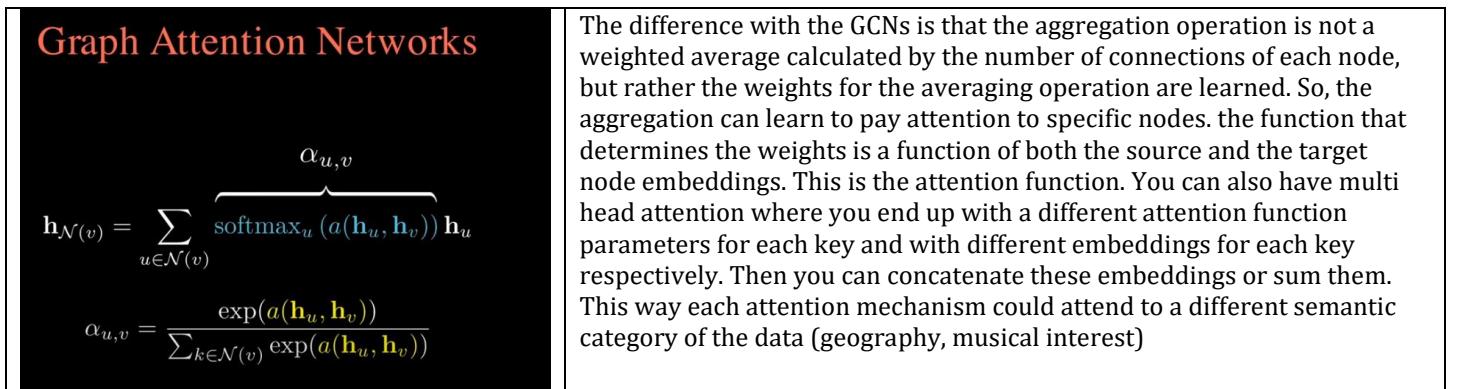
So in a content abuse example in twitter where the tweets are nodes and their shared information is the ips and accounts. But this time accounts and ips are not represented as nodes of the graph. Rather the tweets are connected if they share the same account or ip. We encode the content of the tweet with a vector. We have fraud/no fraud accounts. But if we just used label propagation to characterize a tweet as fraud or not, we will not take the content of the tweet into account. If we just fed the tweet contents (the encoding vectors) to a classifier we will not take the information about the account and the ips. The solution to this is a GCN. Now the content vector is passed along, it is aggregated and passes through a neural network that ends up in a "label" value fraud/no fraud. Because the connection represents that tweets are from same ip or account, all the tweets from the same account or ip are aggregated and the aggregation is classified. This is how the account and ip information is used in addition to the tweet content. We use the information that some tweets are related with each other. So if some of them are fraud (their vector is classified as such) they will influence the others.



Weighted average



GATs (Graph attention networks)



Relational Graph Convolutional networks (RGCNs)

The difference is that RGCNs can handle differently, different types of nodes and types of relationships. In GCNs each layer has one matrix shared by all nodes. In RGCNs each type of connection has its own matrix. So, each layer has r number of W matrices where r is the number of types of relationships (follow, block). This way it can handle differently each relation type which is the correct thing to do.

GCN

\mathbf{a} : adjacency vector

$\alpha_j \doteq 1 \Leftrightarrow v_j \rightarrow v$

$d = \|\mathbf{a}\|_1$: degree (# of incoming edges)

$\mathbf{h} = f(\mathbf{Ux} + \mathbf{VXa}d^{-1})$ $f(\cdot) : (\cdot)^+, \sigma(\cdot), \tanh(\cdot)$

$\{\mathbf{x}_i\}_{i=1}^t \rightsquigarrow \mathbf{H} = f(\mathbf{UX} + \mathbf{VXAD}^{-1})$ $D = \text{diag}(d_i)$

Transformers

In a nutshell

Introduced in 2018. Two important advantages of transformers in relation to RNNs for sequence processing:

1. They can understand larger sequences
2. They are much more efficient to train

In RNNs you must propagate and back propagate through time, and this is a sequential process. In the forward pass the state is processed sequentially, from one timestep to the other. You can't parallelize that. Each state computation requires the result of the previous one. It has temporal dependency. If the sequence is large, the influence of the first timesteps will fade away (vanishing gradients). So you must implement gates (LSTMs) which are still sequential though. In transformers during training, the whole sequence is processed all at once, there are no timesteps where you need the previous result to process the current timestep. The sequences are sets, they have no order fundamentally (you assign the order externally to each item of the set). This allows for parallelization of computations during training. (During inference you kind of have this step by step process where the predicted output becomes the input of the next step). It also means that the output is equivariant to the input. if you change the order of the input the order of output will change in the same way but otherwise it will be unchanged.

Transformers are based to the concept of attention. There is hard and soft attention. In Hard attention you pay attention only to the most important thing, while in soft you consider a range of the most important things. There is self and cross attention. In self attention you have an input x and you decide to which parts of that input to pay attention to, in order to answer questions formulated by x . In cross attention you have an input x and another input ξ and you decide to which part of ξ to pay attention to, in order to answer questions formulated by x .

Attention is created by a (soft)argmax operation which produces a set of scores summing up to 1. The highest the score the more attention is paid to the component associated with that score. The result is a linear combination of the higher score components.

Use cases

- img \mapsto set: image to bounding box (BB) (DETER)
- set \mapsto set: point clouds to BB, surrounding vehicle trajectory pred.
- seq \mapsto seq: translation, conditional image generation (DALL-e)
- seq \mapsto set: event location and duration
- img \mapsto vec: visual image transformer (ViT)
- seq \mapsto vec: movies review

Key value store using attention (Q, K, V)

Transformers implement the concept of key value stores with neural networks, where queries, keys and values are all learned.

Self attention

Queries, keys, and values $\{q_i\}_{i=1}^t \rightsquigarrow Q \in \mathbb{R}^{d \times t}$

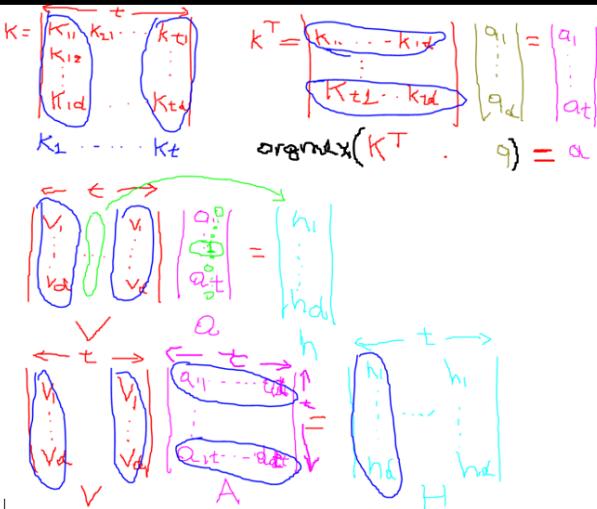
$$q = W_q x, \quad k = W_k x, \quad v = W_v x \quad \beta = \frac{1}{\sqrt{d}}$$

$$q, k \in \mathbb{R}^{d'}, \quad v \in \mathbb{R}^{d''}, \quad d' = d'' \stackrel{\perp}{=} d$$

$$\{x_i\}_{i=1}^t \rightsquigarrow \{q_i\}_{i=1}^t, \{k_i\}_{i=1}^t, \{v_i\}_{i=1}^t \rightsquigarrow Q, K, V \in \mathbb{R}^{d \times t}$$

$$a = [\text{softmax}](\arg\max_\beta(K^\top q)) \in \mathbb{R}^t \quad h = V a \in \mathbb{R}^d$$

$$\{q_i\}_{i=1}^t \rightsquigarrow \{a_i\}_{i=1}^t \rightsquigarrow A \in \mathbb{R}^{t \times t} \quad H = VA \in \mathbb{R}^{d \times t}$$



Lecun mentioned that in self attention the A matrix is normalized per row (softmax of each row).

Self-attention

We have an input x . x is a set of "t" items with no order. We define queries, keys and values as q , k and v vectors which are linear transformations of an item of x (W^*x). we have t items in the x set, so we will have t qs , ks and vs . each of them has length d . So we get the matrices Q , K and V (Q contains the queries for all items of the input set). In the notation used in the picture, q is a query formulated from one item x from the input set.

We construct the attention vector a with the formula $\text{softmax}(K^\top q)$. The intuition is this: Each multiplication of a row of K^\top with the vector q is a dot product between these two vectors. The question vector and this key vector. The dot product though, can be represented by a projection of one to the other and the multiplication of the projection with the other. So the more similar in direction two vectors are, the larger their dot product. This means that the key with the larger similarity with this query will have the highest score. If we have the key we can select the corresponding value. To do so, we multiply the resulting attention vector a , with the values matrix. This operation will select the value that corresponds to that key. For example in hard attention, the resulting vector h (the hidden layer as it is called) will be the most important column of the values matrix. So the h vector is the answer to the q query. The H matrix is the answers to all queries contained in the Q matrix. The system will try to learn (to create) the proper queries, keys and values so that the most similar keys to a query, select a value that matches that query. You can think of this system as a way for the system to learn what items of the input are important, no matter their position within the set.

Cross attention

Queries, keys, and values $\{q_i\}_{i=1}^t \rightsquigarrow Q \in \mathbb{R}^{d \times t}$

$$q = W_q x, \quad k = W_k \xi, \quad v = W_v \xi \quad \beta = \frac{1}{\sqrt{d}}$$

$$q, k \in \mathbb{R}^{d'}, \quad v \in \mathbb{R}^{d''}, \quad d' = d'' \stackrel{\perp}{=} d$$

$$\{\xi_j\}_{j=1}^\tau \rightsquigarrow \{k_j\}_{j=1}^\tau, \{v_j\}_{j=1}^\tau \rightsquigarrow K, V \in \mathbb{R}^{d \times \tau}$$

$$a = [\text{softmax}](\arg\max_\beta(K^\top q)) \in \mathbb{R}^\tau \quad h = V a \in \mathbb{R}^d$$

$$\{q_i\}_{i=1}^t \rightsquigarrow \{a_i\}_{i=1}^t \rightsquigarrow A \in \mathbb{R}^{\tau \times t} \quad H = VA \in \mathbb{R}^{d \times t}$$

Cross-attention

In cross attention the difference is that the keys and values are formulated by a different input set ξ . The queries instead are still formulated from the input set x .

Self attention

Implementation

from the RNN lecture

$$h[t] = g(W_h [x^{[t]}_{h[t-1]}] + b_h)$$

$$h[0] \doteq \mathbf{0}, W_h \doteq [W_{hx} \ W_{hh}]$$

considering h heads we get a vector in \mathbb{R}^{3hd}

$$\begin{bmatrix} q \\ k \\ v \end{bmatrix} = \begin{bmatrix} W_q \\ W_k \\ W_v \end{bmatrix} x \in \mathbb{R}^{3d}$$

$$\begin{bmatrix} q^1 \\ q^2 \\ \vdots \\ q^h \end{bmatrix} = \begin{bmatrix} W_q^1 \\ W_q^2 \\ \vdots \\ W_q^h \end{bmatrix} x \quad \begin{bmatrix} k^1 \\ k^2 \\ \vdots \\ k^h \end{bmatrix} = \begin{bmatrix} W_k^1 \\ W_k^2 \\ \vdots \\ W_k^h \end{bmatrix} x \quad \begin{bmatrix} v^1 \\ v^2 \\ \vdots \\ v^h \end{bmatrix} = \begin{bmatrix} W_v^1 \\ W_v^2 \\ \vdots \\ W_v^h \end{bmatrix} x$$

Number of heads

During the implementation of these attention mechanisms, we stack queries, keys and values in one vector which is the result of multiplying a big matrix with x . this matrix is a stack of W_q , W_k , W_v . Notice that till now, we use one query for each item of the set. But we can formulate h queries per item in the set. We just have q_1, a_2 etc. and multiple query matrices W_{q1}, W_{q2} etc. stacked one on top of the other. The number h is called heads. This is the number of heads (number of queries per item in the set)

A standard attention layer takes as input two sequences X and X' and computes the tensors K , V , and Q as linear functions.

$$Q = w^Q X$$

$$K = W^K X'$$

$$V = W^V X'$$

$$A = \text{softmax}_{\text{row}} \left(\frac{QK^\top}{\sqrt{d}} \right)$$

$$Y = AV$$

When $X = X'$, this is **self attention**, otherwise it is **cross attention**.

Multi-head attention combines several such operations in parallel, and Y is the concatenation of the results along the feature dimension.

Given a permutation σ and a $2d$ tensor X , let us use the following notation for the permutation of the rows: $\sigma(X)_i = X_{\sigma(i)}$.

The standard attention operation is **invariant to a permutation of the keys and values**:

$$Y(Q, \sigma(K), \sigma(V)) = Y(Q, K, V),$$

and **equivariant to a permutation of the queries**, that is the resulting tensor is permuted similarly:

$$Y(\sigma(Q), K, V) = \sigma(Y(Q, K, V)).$$

Consequently self attention and cross attention are equivariant to permutations of X , and cross attention is invariant to permutations of X' .

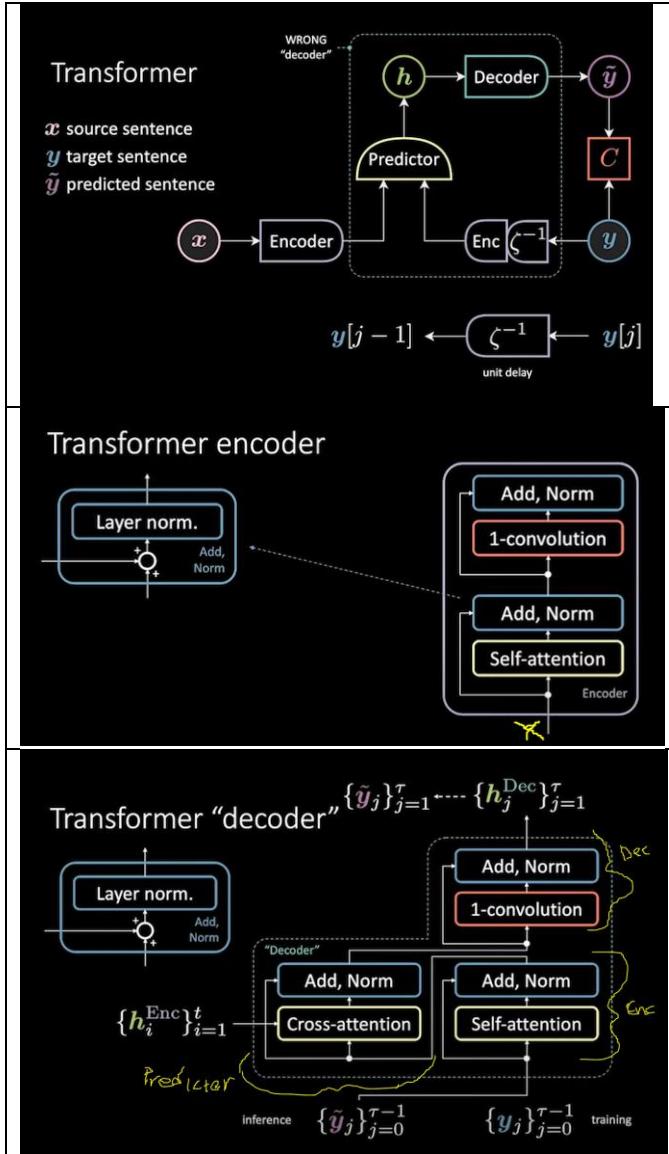
François Fleuret Deep learning / 13.2. Attention Mechanisms

François Fleuret Deep learning / 13.2. Attention Mechanisms

François Fleuret @francoisfleuret · 4h
Don't want to boast, but ain't it a beauty?

Attention representation

The architecture



Have in mind that the original paper describes the whole box as “decoder” which is wrong.

The module ζ is a delay module It is a zeta transformation (from signal processing) which returns the previous item passed through it. If $y(t)$ comes in it will give the $y(t-1)$ that was received in the previous iteration. We can think of this delay as a form of noise, which will make the transformer a delayed DAE. The delayed DAE is called alternatively “language model”. We also have the condition x (the observation x) which makes the transformer a conditioned language model.

In text to text models, x is the input phrase (a set of words). The model tries to predict the phrase that follows. y is the label phrase, the phrase that follows x (y exists only in training). The output y -tilde of the model, represents the predicted phrase. In training the label y becomes the observation x for the next epoch. In evaluation the predicted phrase y -tilde becomes the next observation. This is the **autoregression** concept. The output of the model becomes the next input.

Notice that the dimensionality of y is different from the dimensionality of x (τ and t respectively). This represents the fact that the source sentence can have different number of items (words) from the target sentence.

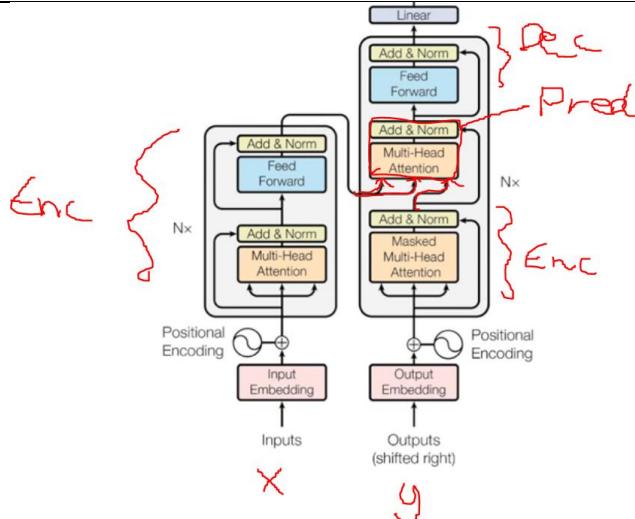
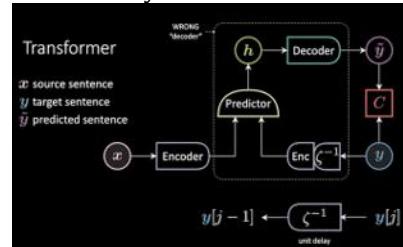
The add and norm operation is a ResNet like approach, probably for preserving the state and the gradient and so being able to have a large number of layers.

In these diagrams you can see how self and cross attention mechanisms are used in a transformer. The encoder of x , creates an embedding h which represents a set of answers to a set of queries formulated by the items of x (kind of represents the important items of x). the same happens with y , then we have a cross attention module where we get the most important items of x based on queries formulated by y (Q is formulated by y , K and V by x).

Example

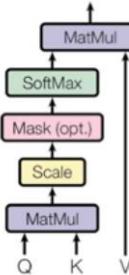
You want to translate the phrase “the cat ate the mouse”. This is the source phrase. (The target phrase y is “η γάτα έφαγε το ποντίκι”). The model produces the predicted phrase word by word. The predicted so far, is fed back to the model. Let’s say we are at the “η γάτα έφαγε”. This is what we have produced so far. Then time t comes. X is the initial phrase, and $y(t)$ is what we will produce now. “η γάτα έφαγε” is the $y(t-1)$, the phrase produced so far. So this is fed to the model and is used for the prediction of the next word. It is used to produce for self attention first, and then as the values V for the cross attention. So the model learns to which part of the source phrase to pay attention and to which part of the phrase that has translated so far, in order to produce the next word. Notice that the whole source phrase and the whole predicted so far phrase are processed by the transformer at once. There is no sequential processing on the inputs. This makes the transformer much more efficient in training than sequential processors like RNNs. The transformer outputs the first word and you can back propagate directly based on this single output, while in an RNN you would have to produce the whole phrase and then back propagate through time. This is the difference.

The diagram on the right is described by this one of Alfredo



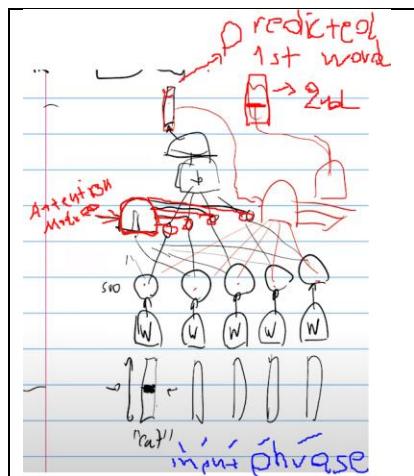
There are three arrows entering the predictor. Q, K from the encoder of x and V from the encoder of the shifted y.

Scaled Dot-Product Attention



BERT 2018 (revolutionary, SSL with transformers - masked text), Roberta 2019

In order for people to fill in the blanks in a sentence, they need to have a model of the world, an understanding of the world. So the reasoning behind masked methods is that in order for the system to correctly fill in the blanks it needs to learn some sort of model of the world. 10-15% of the words of the input are masked. Notice that since the position of words is important in most languages we pass that information to the model. The position for each input item is encoded and added to the input. One positional encoding technique is using sinusoids. A typical transformer has 40 layers of this basic module (self-attention, add and norm). 20 for encoding 20 for decoding. One hot encoded words, are processed by a module to get an embedding for them. The self-attention works on top of these embeddings.



Transformer Architecture

- ▶ Multi-head attention (associative memory)
- ▶ [Vaswani ArXiv:1706.03762]
- ▶ 10 to 60 stages
- ▶ BERT model
- ▶ [Devlin ArXiv:1810.04805]
- ▶ Trained to fill in missing words

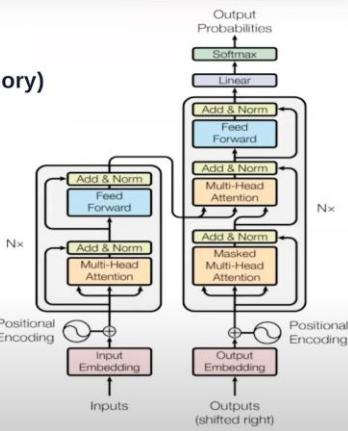
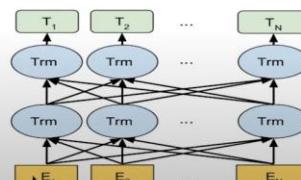


Figure 1: The Transformer - model architecture.

Background

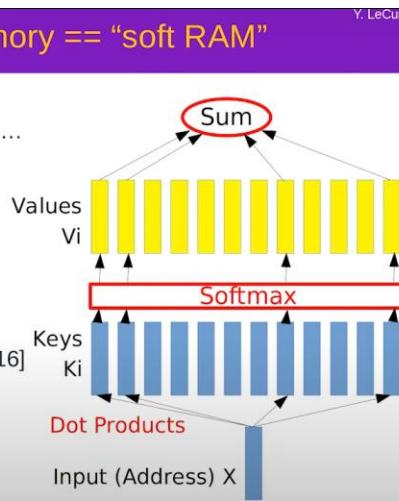
Memory networks (2014)

In the brain long and short term memories are stored in the hippocampus not in the cortex. It has been shown that the state of the cortex becomes independent from its initial state 20 secs ago, which means that we can't remember anything for more than 20 secs using the cortex. The hippocampus is where these things are stored. It has fast changing weights in relation to the cortex. Memory networks is an approach to create systems with memory modules that can be learned.

Differentiable Associative Memory == “soft RAM”

- ▶ Memory Networks, Transformer Network,
- ▶ ELMO, GPT, BERT, GPT2, RoBERTa, XLM-R...
- ▶ Used very widely in NLP
- ▶ Essentially a “soft” RAM or hash table
- ▶ **Memory networks** [Weston et 2014] (FAIR)
- ▶ **Stacked-Augmented Recurrent Neural Net** [Joulin & Mikolov 2014] (FAIR)
- ▶ **Neural Turing Machine** [Graves 2014],
- ▶ **Differentiable Neural Computer** [Graves 2016]

$$Y = \sum_i C_i V_i \quad C_i = \frac{e^{K_i^T X}}{\sum_j e^{K_j^T X}}$$



Transformers are based on what is called a memory network. A model that contains a differentiable associative memory module (a memory that can be learned with gradient based optimization).

The concept is the Q, K, V vectors. In the transformers the keys are parts of the input. in the memory modules, the keys are stored in the memory module.

This memory module is called attention. Maybe not a perfect name.

Applications

Transformers for Language models

You can train transformers on multiple languages and it will eventually learn representations of the meaning of text, independent of language! This means that it could translate phrases from languages it knows little about.

Some applications

- Multi language translation
- Label text (title, sentiment, hate speech, call to violence etc.) pretrained transformer that learns good representations of text. Then trained with supervision for classifying text.
- Solve differential equations
- Chatting about an image

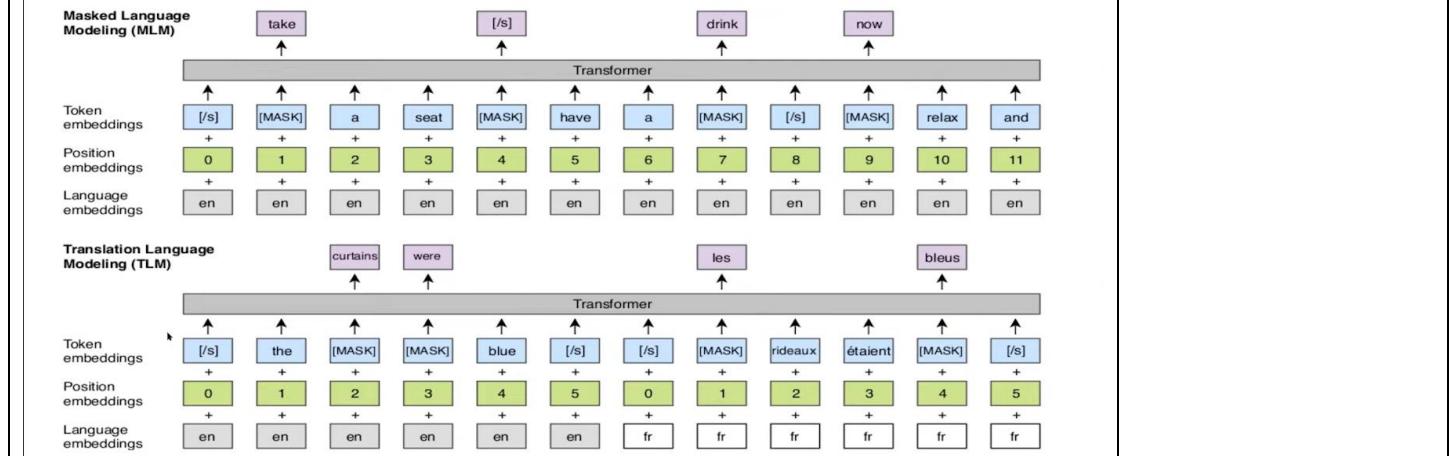
The image is fed to a CNN its embedding to transformer which was trained on human dialogs about images. In this cases it works fine. But it isn't easy to create a practical useful chatbot this way because it is not possible yet, to learn hierarchical sequence of actions. For example you can train a large transformer on Wikipedia. Then ask it a question. What is the capital of Germany? It will answer easily. But if you make a complex question like which country, border with Germany and has the larger trade with China? It will not answer correctly. To do so it has to read tables, sort them, make cross correlations with a different table maybe read a map etc. this is a complex sequence of actions that we don't know hot to learn as of 2021.

Multilingual Transformer Architecture XLM-R

Y. LeCun

Another technique of a transformer for translation

► [Lample & Conneau ArXiv:1901.07291]



Applications of memory networks

You encode phrases and store them in the memory module. For example, john get up from bed. He open the fridge. He picks up a milk. He went to the garden. He saw Jane. The milk fell off his hand. Etc. then you can ask questions to such a system like this: how many people are there? Where at any time two people in the same room? etc. these systems can work well, but they require a lot of data for training. So they are not really practical. They are only used for superficial cases in which you can generate a lot of artificial data.

Transformers for vision

From the DETR paper, the trend is to use CNN + transformers for vision. They can even replace transformers entirely. The concept is that you split the image to patches (overlapping or not) and pass them through a CNN. Then you pass their embeddings to the transformer. Notice that the attention mechanism learns to pay attention to specific parts of the image in order to classify it correctly (προβοσκίδα, ουρά, πόδια για ελέφαντα).

Optimization

1. Gradient free methods (zero order methods)

You might not know the function, so you can't calculate the derivative of it. So you are in the slope of a mountain and you are blind. You get a step, see what changed and go back. Then take a step in another direction and see if the slope changed and how much. Then you can choose to which direction to move. Essentially you estimate the derivative of the cost function by making an action and checking its effect.

In other cases, you might know the cost function, but it might not be continuous or differentiable everywhere. So its derivative doesn't give you any valuable information about the direction you should take. If there is a flat terrain around you, you must take random large steps. Or clone your self in many places and do the same check.

There is the package "nevergrad" which contains a lot of gradient free methods (genetic algos, particle swarm, perturbation methods etc.)

Gradient free optimization methods are used in RL where you don't know the cost function (there is no correct label to get the difference with). You only get a reward which is the output of the unknown cost function but you have no information about the direction that you should choose to increase the reward. There is no correct label to get a difference and back propagate it so that you tune the parameters in the correct direction and with the appropriate size. So you have to estimate the cost function. You can either perturb the parameters of your model to get an estimate of the gradient of the cost function with respect to the parameters. Or you can "perturb" the output of your model to

get an estimate of the cost function with respect to the output of your model and then propagate it through your model with typical backpropagation applying a gradient based optimization method. So in these cases you use both gradient free (perturbation based) and gradient based optimization methods. This was used by Deepmind in their q learning models.

[Forward-forward algorithm Hinton 2022] The idea is to make random perturbations of the weights or the neural activities and to correlate these perturbations with the resulting changes in a payoff function. But reinforcement learning procedures suffer from high variance: it is hard to see the effect of perturbing one variable when many other variables are being perturbed at the same time. To average away the noise caused by all the other perturbations, the learning rate needs to be inversely proportional to the number of variables that are being perturbed and this means that reinforcement learning scales badly and cannot compete with backpropagation for large networks containing many millions or billions of parameters

2. Gradient based methods (first order methods)

3. Second order methods

They are also gradient based methods. They use the second derivative of the cost function to accelerate the process. ADAM and conjugate are kind of in this category. These are rarely practical for ANNs except for some special cases.

Some tips

In gradient based optimization methods of one parameter, there is an optimal learning rate, which can move you directly to the minimum in one step for one-dimension functions (cost functions of one parameter). It is equivalent to the inverse of the Hessian function of the cost function. The hessian function is the second derivative of the cost w.r.t to the parameters. This is the Newton algorithm. When you have more parameters, the step is to the minimum of the largest curvature. But it will not make it faster to converge in the other directions. You have to wait for them to converge. But in ANNs you can't use this method because the hessian is not invertible (the cost function isn't convex in all dimensions, it has places where it goes flat for some dimensions). Also the hessian is not always invertible. It is only if it is positive definite (only positive eigenvalues). If it has negative eigenvalues the function is not convex. There is a method called Leverberg Maquidt for approximating the hessian of a convex function in a way that the approximation is positive definite, but as we said it can't be applied in ANNs. Maybe only in amortized inference in an EBM or a graph model, where you make gradient descent and if the dimensions are small.

The intuition about the eigenvalues of the hessian of the cost and the shape of the cost is that if the cost function is elongated to one direction then its eigenvalue in that direction is larger than the eigenvalue of the other direction. Ideally you want it to not be elongated but this is not possible.

There is a mathematical proof dealing with the hessian matrix which explains why it is necessary to normalize the input x (0 mean and 1 std). it equalizes the curvature of the cost function w.r.t all the parameters (it makes it less elongated). This will make the local hessian matrix of a unit equal to the identity matrix (diagonal with all terms equal to one) so it will be easy to invert.

In the positive definite case (quadratic cost function for example) and for normal gradient descent

- The Learning rate must be smaller than 1 over the largest eigenvalue of the Hessian matrix for gradient descent to converge
- The learning speed is proportional to 1 minus the ratio of the smallest eigenvalue of the Hessian matrix to the largest one. This ratio is called the condition number of the hessian matrix.

Convergence of GD

Y. LeCun

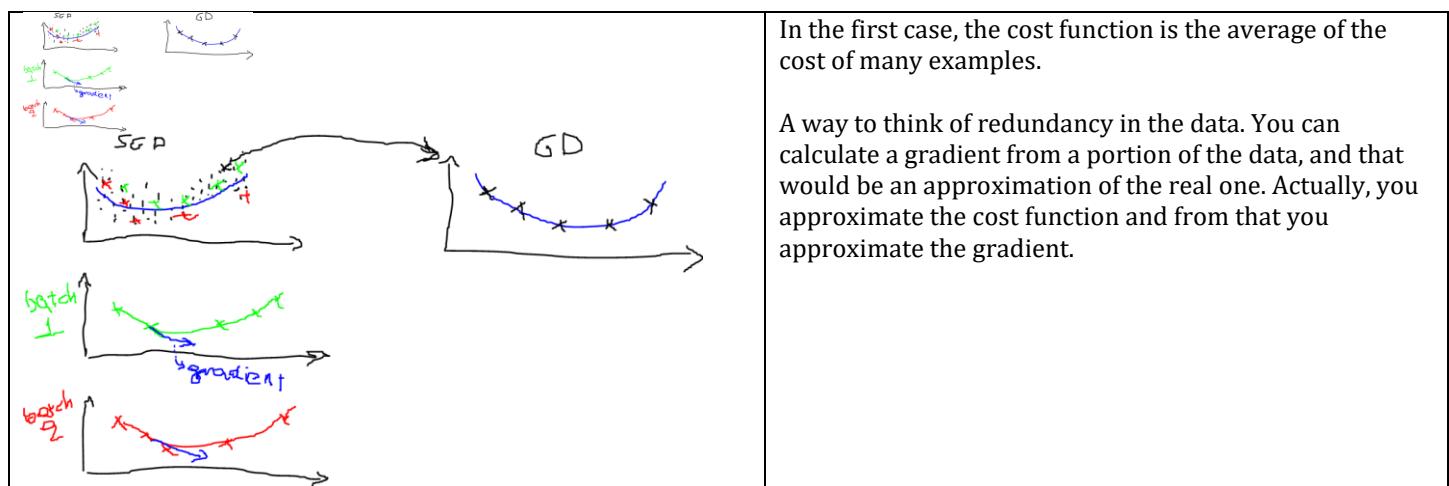
- ▶ Convergence speed depends on conditioning
- ▶ Conditioning: ratio of largest to smallest non-zero eigenvalue of the Hessian
- ▶ How to condition?
 - ▶ Center all the variables that enter a weight
 - ▶ Normalize the variance of all variables that enter a weight

In general in GD converges fast only when you have one learning rate for each dimension.

Stochastic gradient descent

Whenever the objective function (the cost function) is an average of the per sample loss we can use stochastic methods like SGD. We can use SGD when the function we want to minimize is an average of lots of terms. If it is one term only, then we have to use GD. If the function I have is not approximate (not like the per-batch approximation of the dataset loss), then you're performing non-stochastic GD. The stochasticity comes from the approximation to the objective function.

Stochastic gradient descent exploits the redundancy in the data, so in general it is converges faster than gradient descent. If you have a dataset of 10m examples, where 10k examples are repeated 10k times, there is no point in doing GD. You can do GD on the 10k only and this would be 10k times faster. This is what SGD does.

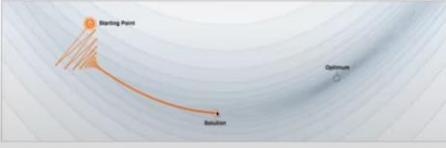
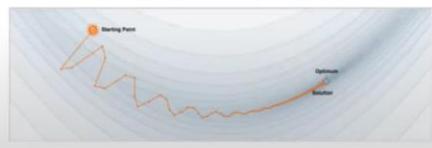


But in amortized inference in EBMs (you have to minimize a cost function -the energy- for one sample. So one sample is all you have and one function) and optimal control (where we have one vector of actions and a very well defined cost function) where the cost function is not the average of a lot of terms, but it is a specific function, we can't use SGD. we either use quasi newton methods which approximate the hessian matrix (they exploit second order properties of the cost) if the num of dimensions is not too large (>100) or conjugate gradient method $O(N)$ time complexity. It exploits the second order properties of the cost function. It requires a line search between different search directions. LBFGS it is a method in between quasi newton method and conjugate gradient method. There is a parameter that defines if you go from one end to the other. N for quasi newton and 1 for conjugate. So you can choose something that suites the specific case. It basically controls the complexity of the matrix that approximates the hessian. Lecun proposes to start with conjugate method in these cases and then if it doesn't work well use LBFGS with a parameter greater than 1. It might be slower than pure conjugate but might converge.

Ideal batch size would be 1! But we use larger batch size because we can parallelize the computations for each sample in GPUs so in aggregate SGD converges faster with batch size larger than 1. But if you increase the batch size a lot, it might be slower to converge than a smaller batch size, because that small size could be enough to exploit the redundancy of the data or in other words it could give a very good approximation of the cost function.

SGD + Momentum

The descent vector is the previous one multiplied by a constant plus the current gradient.

<h3>Momentum acceleration methods</h3> <p>Y. LeCun</p> <p>The most misunderstood idea in optimization?</p> <p><i>gradient</i></p> $p_{k+1} = \hat{\beta}_k p_k + \nabla f_i(w_k)$ $w_{k+1} = w_k - \gamma_k p_{k+1}$ <p><i>Extra term on top of SGD</i></p> <p>SGD + Momentum = Stochastic heavy ball method</p> $w_{k+1} = w_k - \gamma_k \nabla f_i(w_k) + \beta_k (w_k - w_{k-1})$ <p>$0 \leq \beta < 1$</p> <p>This is mathematically equivalent to the previous form for a particular value of beta.</p> <p>Key idea: The next step becomes a combination of the previous step's direction and the new negative gradient</p>	
<h3>Momentum acceleration methods</h3> <p>Y. LeCun</p> $p_{k+1} = \hat{\beta}_k p_k + \nabla f_i(w_k)$ $w_{k+1} = w_k - \gamma_k p_{k+1}$ <p>The optimization process resembles a heavy ball rolling down a hill. The ball has momentum, so it doesn't change direction immediately when it encounters changes to the landscape!</p> <p>Without momentum</p>  <p>facebook Artificial Intelligence Research</p> <p>With momentum</p>  <p>Image Source: https://distill.pub/2017/momentum/</p>	<p>It helps a lot with the condition issue (elongated cost functions where some dimensions are steeper than others).</p> <p>Notice though that it might increase the convergence speed but it might end up in a solution that is worse in relation with not using it. There might be cases where you want the noise at the end, instead of the smoothing produced by momentum.</p>

Adam

Adam

Y. LeCun

Adam: RMSprop with a kind of momentum

"Adaptive Moment Estimation"

Presented here without **bias-correction** (i.e. steady state Adam)

Momentum (Exponential moving average)

2nd moment estimate
(same as RMSProp)

$$m_{t+1} = \beta v_t + (1 - \beta) \nabla f_i(w_t)$$

$$v_{t+1} = \alpha v_t + (1 - \alpha) \nabla f_i(w_t)^2$$

$$w_{t+1} = w_t - \gamma \frac{m_t}{\sqrt{v_{t+1}} + \epsilon}$$

Use m instead of the gradient

Just as momentum improves SGD, it improves RMSProp as well.
The exponential-moving-average method of updating momentum is equivalent to the standard form under rescaling. Nothing mysterious.

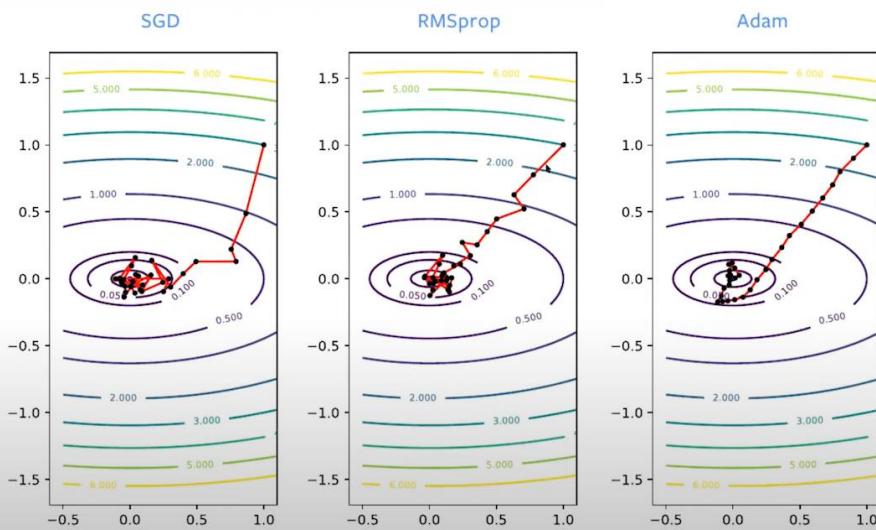
The full version of Adam has **bias-correction** as well, which just keeps the moving averages **unbiased** during early iterations. The algorithm quickly approaches the above steady state form.

Diederik, Kingma; Ba, Jimmy (2014). "Adam: A method for stochastic optimization"

In RMSprop essentially you normalize the gradient vector by which you update the parameters. You divide the exponential moving average of the gradient for a specific weight, with kind of the standard deviation of the gradients of that weight (the sqrt of the square gradient). So the gradient is normalized. Adam is similar. It is being done for each parameter (each weight)

Pure SGD vs RMSprop vs Adam

Y. LeCun



Practical considerations

Y. LeCun

For poorly conditioned problems, Adam is often **much** better than SGD.
I recommend using Adam over RMSprop due to the clear advantages of momentum

BUT, Adam is poorly understood theoretically, and has known disadvantages:

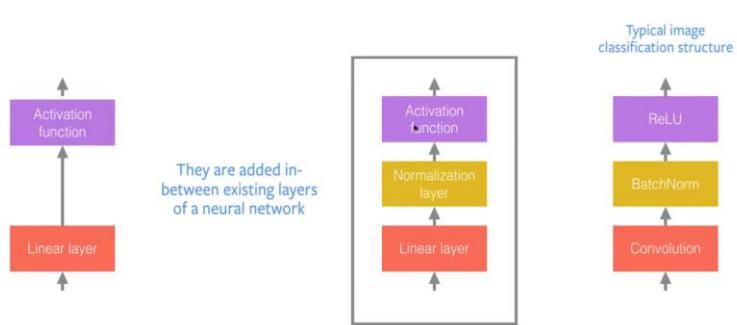
- Does not converge at all on some simple example problems!
- Gives worse generalization error on many computer vision problems (i.e. ImageNet)
- Requires more memory than SGD
- Has 2 momentum parameters, so some tuning may be needed

It's a good idea to try both SGD + momentum and Adam with a sweep of different learning rates, and use whichever works better on held out data

Normalization techniques

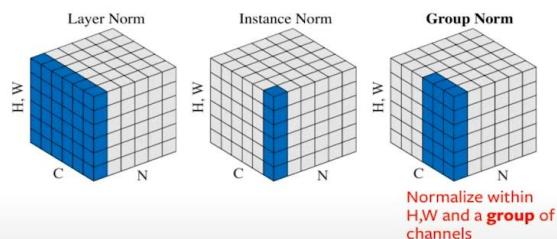
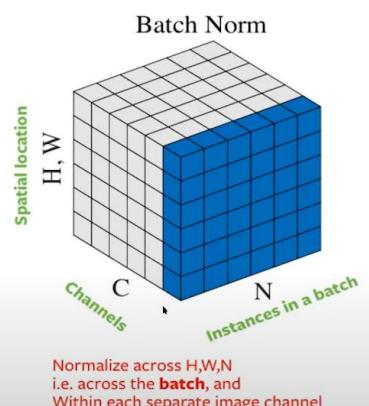
Normalization layers

Y. LeCun



Types of normalizations

Y. LeCun



In practice group/batch norm work well for computer vision problems, and instance/layer norm are heavily used for language problems.

IMG SOURCE: <https://arxiv.org/pdf/1803.08494.pdf>

Batch norm for vision

Instance/layer norm for language

Layer norm: averages over space and channels but not over time (instances of a batch)

Diffusion models todo

https://huggingface.co/blog/stable_diffusion#how-does-stable-diffusion-work

Stable diffusion

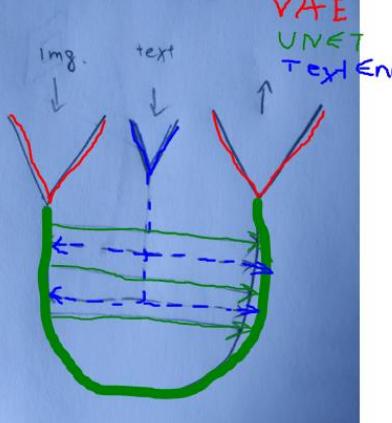
Having seen the high-quality images that stable diffusion can produce, let's try to understand a bit better how the model functions. Stable Diffusion is based on a particular type of diffusion model called Latent Diffusion, proposed in [High-Resolution Image Synthesis with Latent Diffusion Models](#).

Generally speaking, diffusion models are machine learning systems that are trained to *denoise* random Gaussian noise step by step, to get to a sample of interest, such as an *image*. For a more detailed overview of how they work, check [this colab](#).

Diffusion models have shown to achieve state-of-the-art results for generating image data. But one downside of diffusion models is that the reverse denoising process is slow because of its repeated, sequential nature. In addition, these models consume a lot of memory because they operate in pixel space, which becomes huge when generating high-resolution images. Therefore, it is challenging to train these models and also use them for inference.

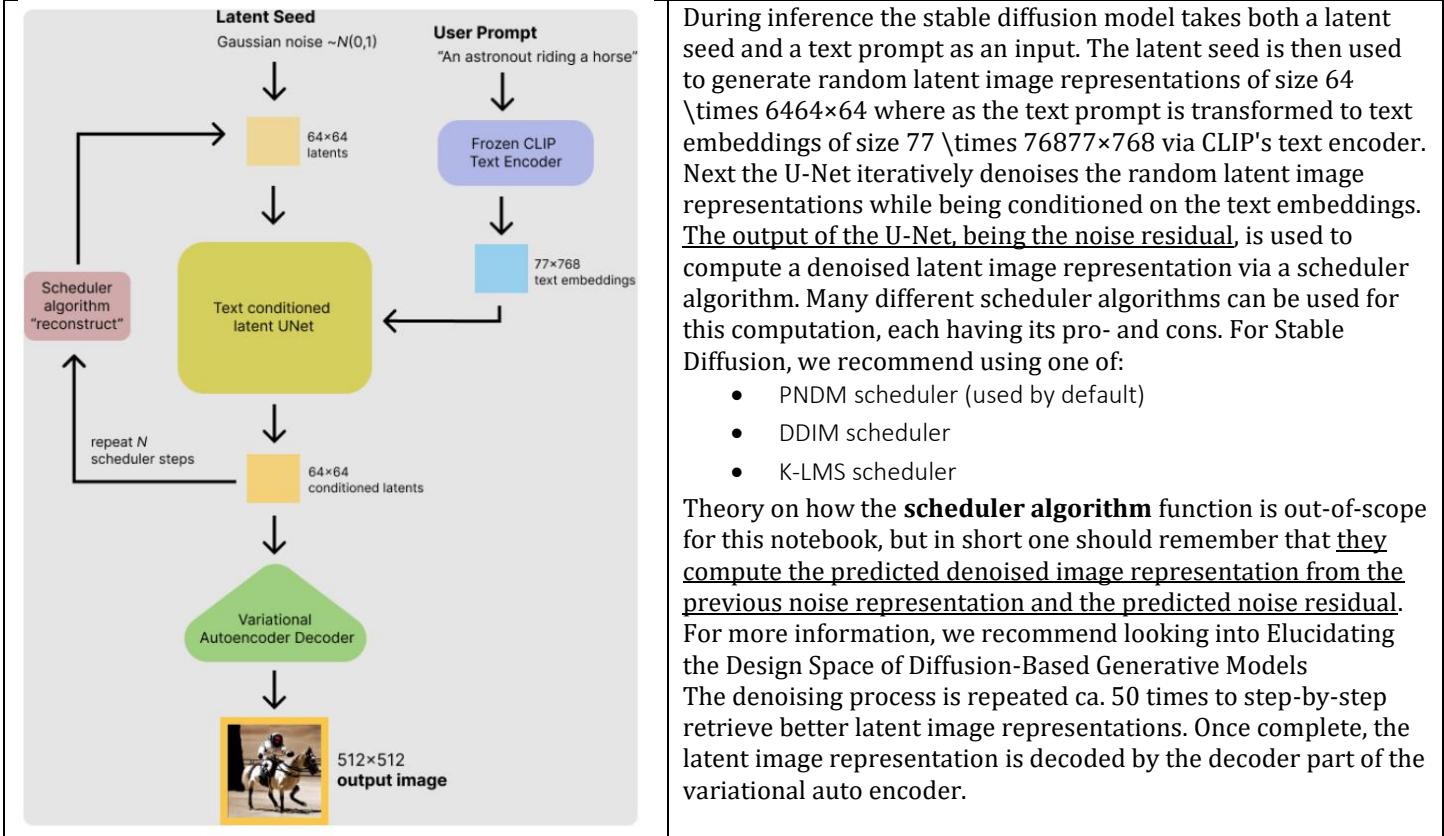
Latent diffusion

Latent diffusion can reduce the memory and compute complexity by applying the diffusion process over a lower dimensional *latent* space, instead of using the actual pixel space. This is the key difference between standard diffusion and latent diffusion models: [in latent diffusion the model is trained to generate latent \(compressed\) representations of the images](#).

	<p>There are three main components in latent diffusion.</p> <ol style="list-style-type: none">1. An autoencoder (VAE).2. A U-Net.3. A text-encoder, e.g. CLIP's Text Encoder.
--	---

Since latent diffusion operates on a low dimensional space, it greatly reduces the memory and compute requirements compared to pixel-space diffusion models. For example, the autoencoder used in Stable Diffusion has a reduction factor of 8. This means that an image of shape $(3, 512, 512)$ becomes $(3, 64, 64)$ in latent space, which requires $8 \times 8 = 64$ times less memory. This is why it's possible to generate 512×512 images so quickly, even on 16GB Colab GPUs!

Inference



Stable Diffusion

<https://yang-song.github.io/blog/2021/score/>

<https://github.com/johnowhitaker/aiaiart>

Diffusion-LM shows strong performance in controllable generation, but it remains an open question whether it could match autoregressive LMs in PPL and speed.

Misc

Exploding/vanishing gradients

Exploding gradients

Imagine that you have a NN with multiple layers but only one unit in each layer. You have ReLUs in all layers. Assume all weights are 1. You have a positive input let's say 1. 1 is multiplied by the weight which is 1, and passes through the relu which gives also 1. So you get 1s in the whole network till the end. The same happens in the backpropagation step too. if the gradient is 1 it backpropagates and reaches 1 at the first layer. Assume all weights are all 2. In the first layer the activation value is 2, then 4 then 8 and so on. This is an **exploding state**. This explosion happens also in the backpropagation step. Let's say the final gradient is 1. It becomes 2 for the second last layer, then 4, then 8 etc. so you have an **exploding gradient**. This situation is bad because it means that the weights of the lower layers will change a lot, while the weights of the higher layers a little (you need a large change to the lower layers to change the higher layers even slightly). This is problematic. The state amplitudes

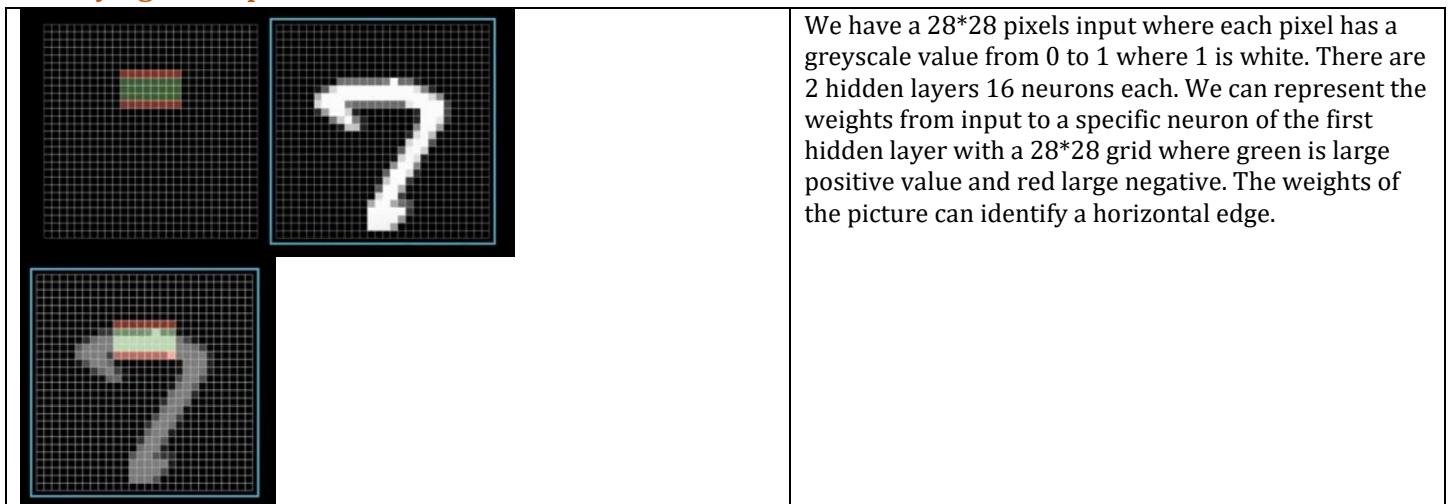
must be more or less the same and the gradient amplitudes too. This is the reason why normalization techniques like batch or group normalization are used (for the state and the gradient I suppose). This way the activations and gradients are all with (0 mean?) variance of 1 approximately. Applying Softmax is a normalization technique.

An advantage of softmax for classification of mutually exclusive classes is that it makes the output units competing with each other in the sense that if one gets bigger some others should become smaller because the sum must be 1. This is exactly what we want for this type of classification problems.

Vanishing gradients

This is the opposite problem. If you use activation functions that have regions with very small or no gradience at all and the network goes to activations in that region then the gradients become 0 and you can't back propagate so you don't know how to change the weights. I guess that normalization can solve this problem too.

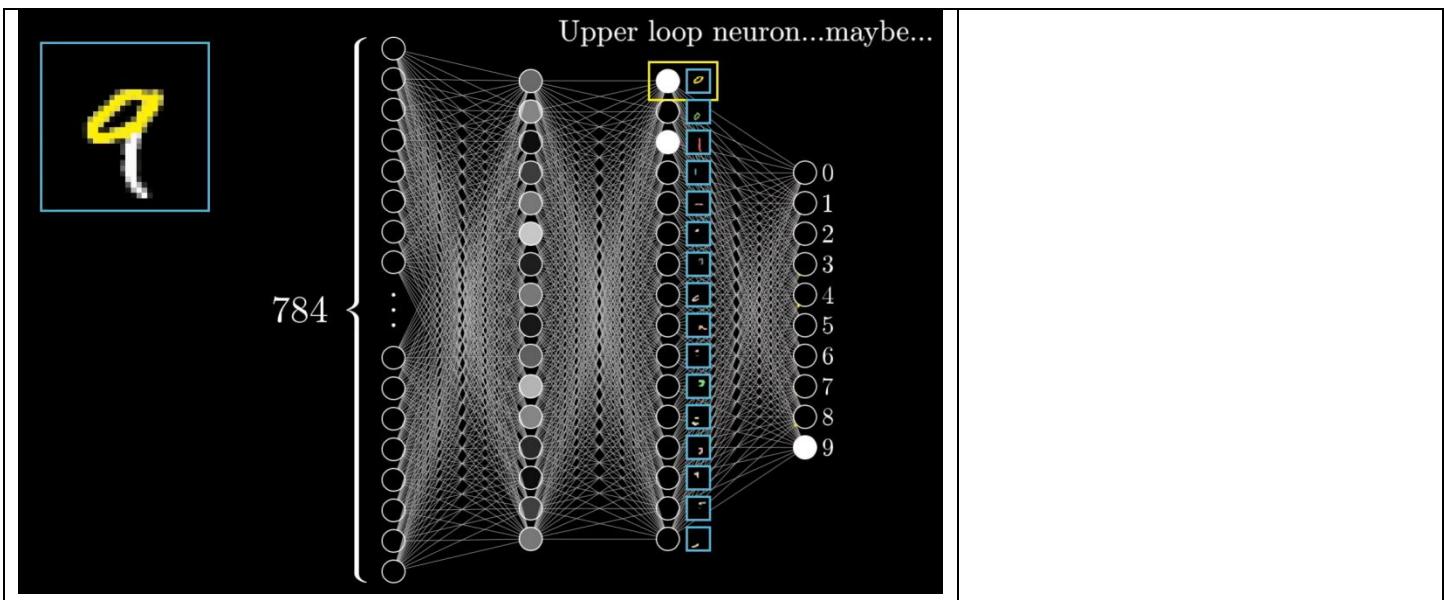
Identifying visual patterns



This neuron will be activated if there is a horizontal edge on that part of the picture. We need the negative weights too. imagine a fully white input. The sum of large positive weights times large input and large negative weights times large input will give an output close to 0 and the neuron will not be activated. But in the input of the image, the positive weights are multiplied by large input values and the negative weights with very small or 0 input so the weighted sum is large and the sigmoid (Relu actually) is activated.

Intuition on visual patterns a neuron might learn

Have in mind that a specific pattern (like a logical AND or a loop like shape) is represented by a single neuron, but if we want to represent this pattern with an embedding (a representation set), we just take the activations on the previous layer.



	<p>The patterns of the second hidden layer could potentially be formed by some simpler patterns (edges) in the first hidden layer.</p>
--	--

Notice that this is just a hypothesis on how this ANN could learn to recognize digits. Actually it is an optimistic hypothesis because if it learns these patterns then it has indeed learned properly and could generalize on unseen examples, it hasn't just memorized the training set (or in other words it didn't overfit to the training set).

What second layer neurons look for

What???

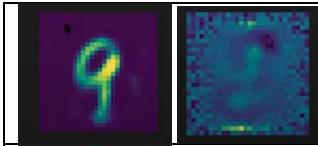
After training this multilayer perceptron we got a success rate of 98% (state of the art was 99.97%). These are the weights of the first hidden layer of the trained network. We hoped to see some patterns here like small edges that will form larger patterns in the next layer but this is not the case. We just have random "patterns". This is an indication that our network didn't learn so well. It only saw centered images of some digits after all (not miscentered images, without trying to recreate parts of the image etc.).

You can confirm that it hasn't really learned well if you give it a random image. It will classify it with confidence as a specific digit instead of showing uncertainty.

	<p>Notice that the patterns would have appeared in the <u>plot of the weights</u>. For example, this pattern of weights for a neuron would recognize an edge at that specific place.</p>
--	--



These gradients identify low frequency regions (same pixels) on the background of the images. If you have a black region (all 0s) then this gradient will give a non-zero value and essentially you detect that low frequency region.
If the weights were all zero, it would just ignore the low frequency area. Not learn anything from it.



The area outside of the digit, has low spatial frequency meaning it is a uniform region of pixels (with value 0 or very close to it in this case).
A typical autoencoder will learn to generate salt and pepper noise for these areas (second picture) so that their sum is cancelled out.



A DAE on the other hand, learns to ignore what happens in these low frequency regions. The reason is the noise added to the input. now these regions are not of low frequency they have some random patterns that don't contribute to the cost though. So it leans to ignore them completely adding 0 weights in these areas.

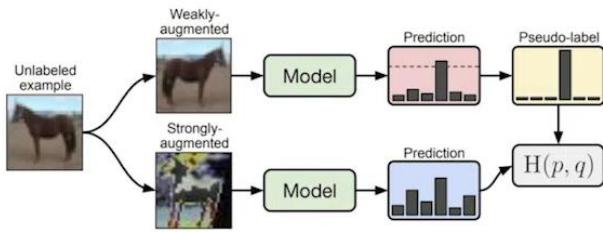
Semi-supervised methods

They include various Pseudo label techniques. You must find clever ways to select which data to pseudo-label. You must select the examples your model is the most confident about (in contrast in active learning techniques where you select which data to label to fine tune your model with it, you want to select the examples your model is the least confident about). Then you use those pseudo-labels to further train your model with.

You have a large unlabeled dataset. A small part of it is labeled. You train your model in a supervised manner in the small dataset. Then you use that trained model to classify the unlabeled data. Then you use these predictions as labels (pseudo-labels) and retrain the model to the whole dataset in a supervised manner. You might choose to pseudo-label only a portion of the data, for which the model performs best. Then retrain it with them too. and repeat the process a few times.

FixMatch and CoMatch are two standard approaches.

FixMatch



$$\ell_s = \frac{1}{B} \sum_{b=1}^B H(p_b, p_m(y | \alpha(x_b)))$$

$$\ell_u = \frac{1}{\mu B} \sum_{b=1}^{\mu B} \mathbb{1}(\max(q_b) \geq \tau) H(\hat{q}_b, p_m(y | \mathcal{A}(u_b)))$$

Active learning

As I understood from the lecun's class projects, it is a way to select which examples to use for the supervised fine tuning of a pretrained SSL model. In general, you want to select the examples the model is most confused about. There are various techniques for this: core set, pool based sampling, margin base sampling, highest entropy. The winner team on the course project labeled 0.5% of the data.

Labeling Request

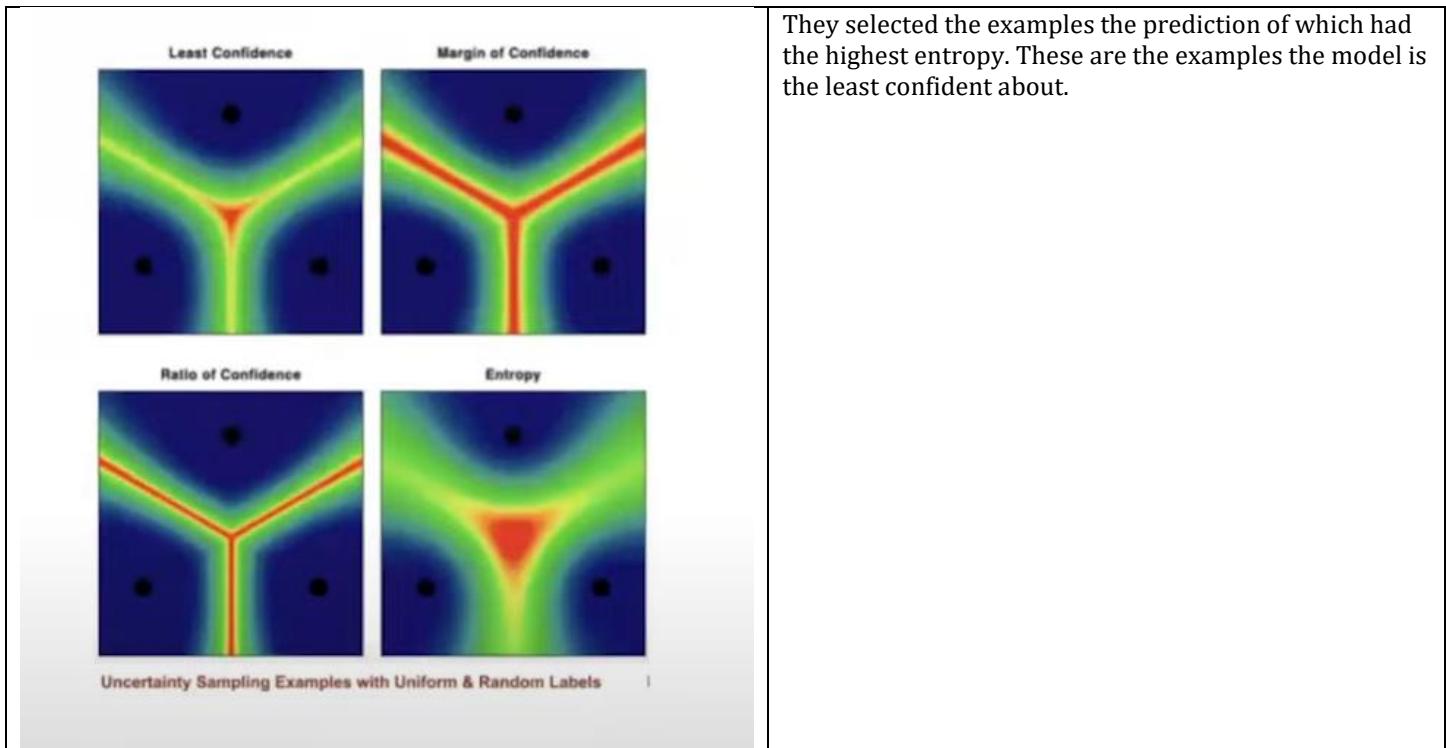
Active learning - Clustering using margin based sampling

- Margin used was the difference in top predicted classes of each image
- Cluster images that are similar, and take the top k images (from each cluster) that the model is confused about

Evaluated with Barlow, Swav and BYOL, and took the intersection among the three models.

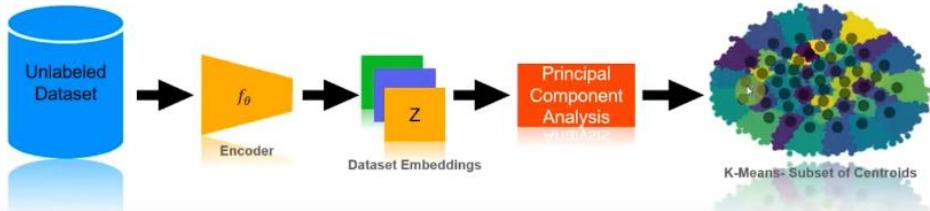
margin base sampling
A nice technique according to Lecun.

the margin used is the difference between the 2 top predicted class for an image. For example, an image is predicted with 0.8 as A and 0.1 as B and 0.1 all other classes. The margin is 0.7 in this case. You pick the ones with the smallest margin which means that the model is not so sure about. You perform k-means on them. Split them to 800 clusters then chose 16 of each cluster. They labeled those and fine tuned the model with them.



Active Learning Core-Set Approach

- Core-Set → Chooses set of points such that a model learned over the selected subset is competitive for the remaining data points.



- Run K-means on compressed embeddings for 12,800 clusters.
- Black circle represents the closest image to the centre point (most representative data point) of that cluster.

Ozan Sener and Silvio Savarese. Active learning for convolutional neural networks: A core-set approach. In International Conference on Learning Representations, 2018.

Core set

In that project they used it to select a subset of images to fine tune the model with (using their labels).

Priors in DL

According to Lecun, the best bet on a way to add priors in the form of rules, or facts to DL systems, is with some sort of large scale associative memories (or information retrieval). You can have a set of rules in the form of text. You can embed them in a vector space. When your system needs to use them it can retrieve them with **associative retrieval** (soft associative memories). Transformers use this to some extend. Dialog systems use these kinds of large scale associative memories as knowledge bases.

Lecun:

Inductive biases are often based on assumptions of symmetry.

- Transformers: equivariance to permutations.
- ConvNets: equivariance to translations.

The no-free-lunch theorems tell us that, among all possible functions, the proportion that is learnable with a "reasonable" number of training samples is tiny.

Learning theory says that the more functions your model can represent, the more samples it needs to learn anything

Consequence: the more priors you put in, the fewer samples you require.

But: the more priors you put in, the greater the chance that the functions you need to learn are not realizable (or hard to learn) by your model.

Tools

Tools for building ML models

Predibase (2022, \$16m, USA)

Low code declarative tool

Resources

<https://tutobase.com/t/MachineLearning?tag=course> a lot of sources for various topics of DL

https://www.youtube.com/watch?v=zQY2YvkMbHI&list=PLkRLdi-c79HKEWoi4oryj-Cx-e47y_NcM&index=1 Ng videos

<https://www.deeplearning.ai/> Ng resources

<https://atcold.github.io/NYU-DLSP21/> deep learning course 2021 for New York University, Yann Lecun,

<https://www.youtube.com/watch?v=mTtDfKgLm54&list=PLlHTzKZzVU9e6xUfG10TkTWApKSZCzuBI> the same in youtube

<https://www.youtube.com/watch?v=vT1JzLTH4G4&list=PLSVEhWrZWDHQTBmWZufjxpw3s8sveJtnI> Stanford CS231n

have also in mind these recommendations from 2016 from Open AI co founder

If you'd like to take courses... Pieter Abbeel and Wojciech Zaremba suggest the following course sequence:

- Linear Algebra — Stephen Boyd's EE263 (Stanford)
- Neural Networks for Machine Learning — Geoff Hinton (Coursera)
- Neural Nets — Andrej Karpathy's CS231N (Stanford)
- Advanced Robotics (the MDP / optimal control lectures) — Pieter Abbeel's CS287 (Berkeley)
- Deep RL — John Schulman's CS294-112 (Berkeley)

If you'd like to get your hands dirty... Ilya Sutskever recommends implementing simple MNIST classifiers, small convnets, reimplementing char-rnn, and then playing with a big convnet. Personally, I started out by picking Kaggle competitions (especially the "Knowledge" ones) and using those as a source of problems. Implementing agents for OpenAI Gym (or algorithms for the set of research problems we'll be releasing soon) could also be a good starting place.

Pytorch

https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html

<https://www.youtube.com/watch?v=GIsg-ZUy0MY> (10 hours)

<https://www.youtube.com/watch?v=c36lUUr864M> zero to intermediate (4,5 hours)

https://www.youtube.com/watch?v=OMDn66kM9Qc&list=PLaMu-SDt_RB5NUm67hU2pdE75j6KaI0v2&index=1 pytorch + lightning (alfredo + lightning creator)

<https://pytorch.org/tutorials/>

1. https://www.youtube.com/watch?v=gUF6WUq0Cf4&list=PLaMu-SDt_RB6-e7GJRQ6cAssjMizOOZUP PyTorch
Lightning Training Intro
2. https://www.youtube.com/watch?v=OMDn66kM9Qc&list=PLaMu-SDt_RB5NUm67hU2pdE75j6KaI0v2 PyTorch
Lightning Masterclass (alfredo too)

<https://towardsdatascience.com/understanding-pytorch-with-an-example-a-step-by-step-tutorial-81fc5f8c4e8e> a good step by step tutorial

Misc

Knowledge distillation

Knowledge distillation aims at leveraging the dark knowledge of a teacher network to improve the performance of a student network with fewer parameters.

A common problem with current DL models is that they don't work well when the test data are in a different distribution from the training data. Some possible approaches to deal with this problem are

Domain adaptation

Transfer learning

Human fps

By the time a human child turns 5, she has seen the equivalent of 800 million "frames" of video + audio + touch to learn how the world works through Self-Supervised Learning.

Much of it is acquired actively.

5 years * 365 days * 12 hours * 3600 seconds * 10 fps = 788.4 million

Inpainting algorithms

From cv2 import inpaint, INPAINT_NS, INPAINT_TELEA

These are SOTA algos.

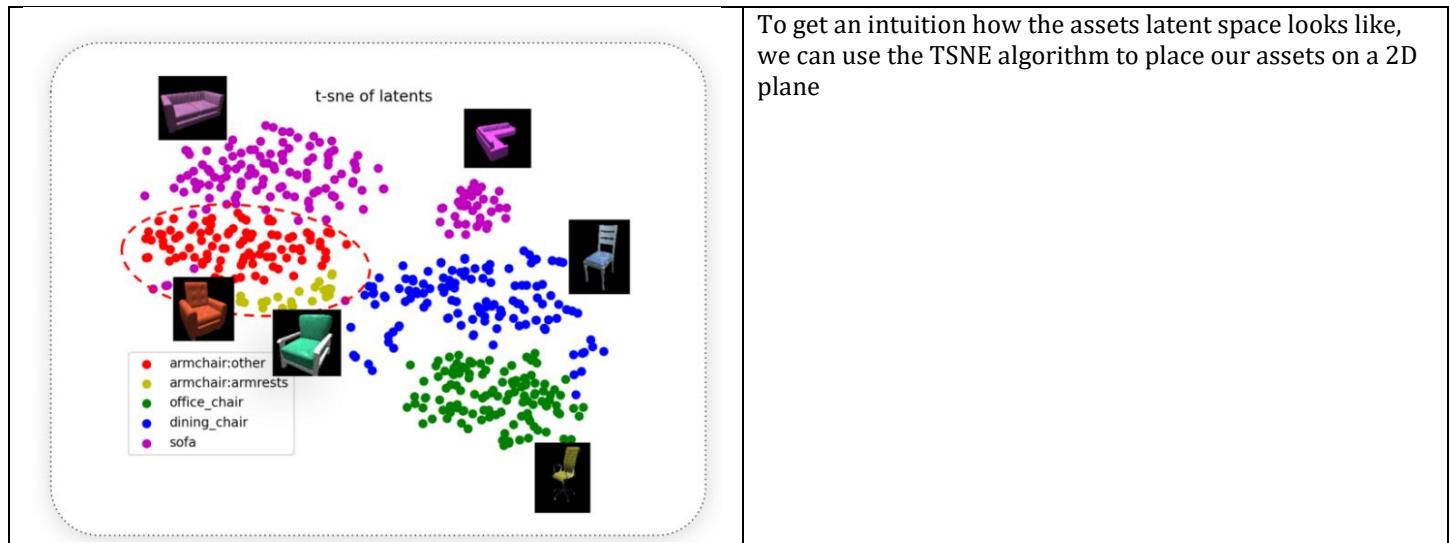
Generative ANNs

With diffusion models outperforming the conceptually difficult GANs, is it time to stop teaching GANs?

Key takeaways (or cons of main models):

- GANs: unstable training, low diversity
- VAE: surrogate loss
- Flow-based: special architecture for reversible transform
- Diffusion: slow to sample

Tsne algo



A cat

900m neurons

10^{13} synapses (10 trillion)

Only a factor of 10 larger than the largest current models.

(assuming 1 synapse is assimilable to 1 parameter).

Planning and Control

Control theory intro

In a nutshell

In general, control is an optimization problem and thus the term optimal control. It refers to the process of finding the optimal parameters of the controller (for example the matrix K in typical linear systems) via an optimization process where we minimize a cost w.r.t to the parameters of the controller, constrained by the dynamics of the system.

Control problems can be classified in 2 large categories. Control of linear systems (or nonlinear systems that can be linearized) and highly nonlinear or completely unknown systems. Classical linear theory of control works for linear systems, systems where the dynamics of the system can be described with linear differential equations, for example \dot{x} is a linear function of x . It also works for non-linear systems that we can be linearized under certain conditions (around a state). For all the other cases where the system is highly nonlinear and can't be linearized around a state, or if it is completely unknown, we apply other methods that fall under the umbrella of data-driven control (or ML control). In these cases ML (or DL) can be used to 1. Create a data driven model of the world for example approximating the unknown function f with ML techniques, or with DL models. 2. Learn controllers, for example learn a DL model that generates the proper control actions and 3. To learn good sensors and actuators, for example where to place the sensors and the actuators. Data driven control is an active area of research.

In linear systems (linear quadratic regulator, Kalman filter, LQG), we have a linear optimization problem which gives us a closed form solution. (as I understood it from the overview only it means that we can find a set of optimal parameters for the controller and then the controller operates with them). But in nonlinear systems the optimization becomes really nasty, and we can't find those closed form solutions. Instead, we are forced to do optimization on the fly like in the model predictive control approach, where we continuously do an optimization for a window ahead in time (for a few timesteps) as we operate the controller.

The car control case shown by Yann Lecun is a data driven control approach, specifically predictive model control. Deep RL is another data driven control approach (where the cost is given at the end of a long sequence of actions instead of in every action so model predictive control can't be applied). Although these methods have limitations. Lecun says that we don't yet have good planning techniques because we don't know how to plan complex sequence of actions or how to create hierarchical representations of action plans.

Linear control theory

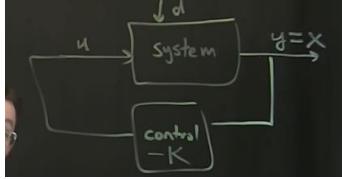
1. <https://www.youtube.com/watch?v=Pi7l8mMjYVE&list=PLMrJAhleNNR20Mz-VpzgfQs5zrYi085m>

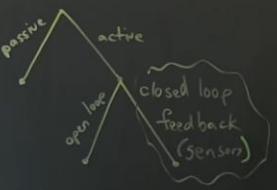
Data driven control

2. https://www.youtube.com/watch?v=ouLR06lj_E&list=PLMrJAhleNNQkv98vuPjO2X2qJO_UPeWR

(https://www.youtube.com/watch?v=oBc_BHxw78s&list=PLUMWjy5jgHK1NC52DXXrriwhVrYZKqjk another good source)

Linear control theory

	$\dot{x} = Ax$ $x(t) = e^{At} x(0)$	X is a vector describing the state of a system. This is a system described with a linear system of equations $\dot{x} = Ax$. If A has any eigenvalue with positive real part then the system will be unstable. If all real parts are negative then the system has stable dynamics, it goes to 0 as time goes to infinity.
---	-------------------------------------	--



$$\dot{x} = Ax + Bu$$

$$y = Cx$$

$$\dot{x} = Ax - BKx$$

$$= (A - BK)x$$

↓ d
System
control
-K

Why feedback?

1. Uncertainty
2. Variability
3. Disturbances
4. Efficient

If we add control u into the system it is now described by $\dot{z} = Ax + Bu$ and we actually end up with a new linear system $\dot{x} = (A - BK)x$

Not all linear systems can be controlled to a stable state. And not all controllers can control a linear system to a stable state.

There are ways to check if a system can be controlled to stable state.

$x(t) = T e^{Dt} T^{-1} x(0)$

We can calculate the eigenvector and eigenvalue matrices and simplify our equations. T is the eigenvector matrix and D the diagonal eigenvalue matrix.

When there is an eigenvalue with a positive real part the system explodes, the x state goes to infinity. If we calculate the eigenvalues of our system and some of them are a bit positive, then we can try to move them into the negative area by introducing the controls to the system. This way we define a new linear system where the matrix A is now replaced with $(A - BK)$ so this new matrix has different eigenvectors and eigenvalues.

Data driven control

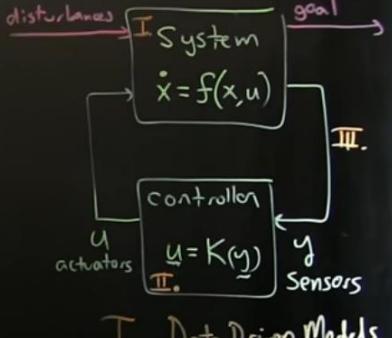
Data-Driven Control
Machine Learning

Challenges:

- Nonlinear
- Unknown dynamics
- High Dimensional
- Limited measurements

What is control?
Optimization constrained by dynamics.

What is ML?
Powerful NL opt. based on data.



I. System: $\dot{x} = f(x, u)$
II. controller: $u = K(y)$
III. Sensors: y
Actuators: u

An overview

Planning in DL Intro

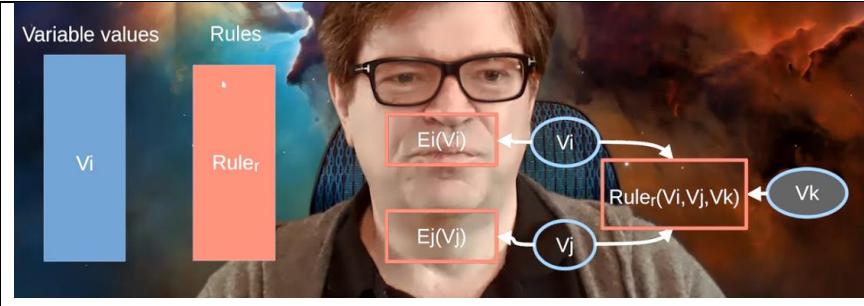
A big challenge according to Lecun is learning hierarchical action plans. Doesn't know how to do it as of 2021.

System 1 type of reasoning is reactive reasoning. System 2 is planning reasoning.

Lecun frames reasoning as a constraint satisfaction or energy minimization problem. Examples of that are the following:

- Probabilistic logical inference
- Optimal control
- Transformers (Attention mechanism)

Probabilistic logical inference



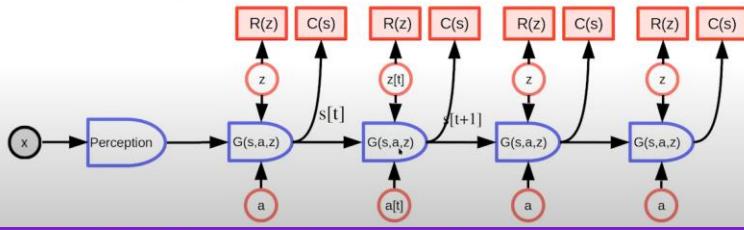
You have several stored variables that represent information about the environment. (a truck hit the house, an earthquake took place etc.). Some are known some unknown. You have a set of rules. A rule is something that gets a subset of all variables and produces values for some other variables. You take each rule and apply it. Some variables take values, so they become known. You run the next rule and the next one and so on, until the values of the variables converge.

When the rules and variables are predefined, this is a traditional symbolic AI approach. But you could learn the rules with DL. This is a research area, not much have been done as of 2021.

Optimal Control (Planning a sequence of actions)

Forward Model for Model-Predictive Control

- ▶ Forward model: $s[t+1] = G(s[t], a[t], z[t])$
- ▶ Cost/Energy: $f[t] = C(s[t])$
- ▶ Latent variable z sampled from $q(z)$ proportional to $\exp(-R(z))$
- ▶ Optimize $(a[1], a[2], \dots, a[T]) = \operatorname{argmin} \sum_t C(s[t])$
through backprop (== Kelley-Bryson adjoint state method)



Y. LeCun

This is not RL, it is optimal control (model-predictive control specifically). In RL we don't know the cost function. The cost (or reward) comes from the world at the end of a sequence of actions. We don't have one value for each action like we have when we know the cost function. This is the problem of sparse rewards. The reward is sparse, so training RL systems is difficult. It requires a lot of repetitions. Here we want to minimize the sum of the cost for all actions. In terms of RL this would be model based RL system, because the state of the world is predicted by a model, based on the previous state and taken actions.

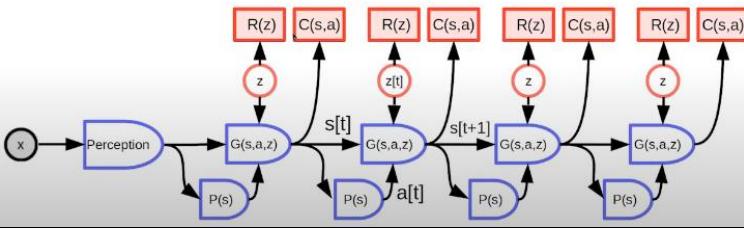
Notice that instead of inferring the sequence of actions that minimize the cost, we can use a model that learns to predict the action (a policy model). This system can be trained end to end with backprop and actually works. This way we learn a controller.

This approach has been tested with the car example, where we want to predict the positions of the cars around us.

They used a conditional VAE for predicting the new state. (so it is the G module of the action predicting system). Notice the

Forward Model for Gradient-Based Policy Learning

- ▶ Forward model: $s[t+1] = G(s[t], a[t], z[t])$
- ▶ Cost/Energy: $f[t] = C(s[t], a[t])$
- ▶ Latent variable z sampled from $q(z)$ proportional to $\exp(-R(z))$
- ▶ Policy: $a[t] = P(s[t])$
- ▶ Learn P through backprop (== Kelley-Bryson adjoint state method)



Y. LeCun

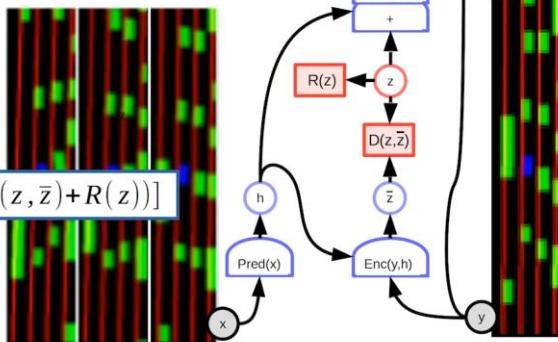
VAE + Drop Out

► Training:

- Observe frames
- Compute h
- Predict \bar{z} from encoder
- Sample z , with:

$$P(z|\bar{z}) \propto \exp[-\beta(D(z, \bar{z}) + R(z))]$$

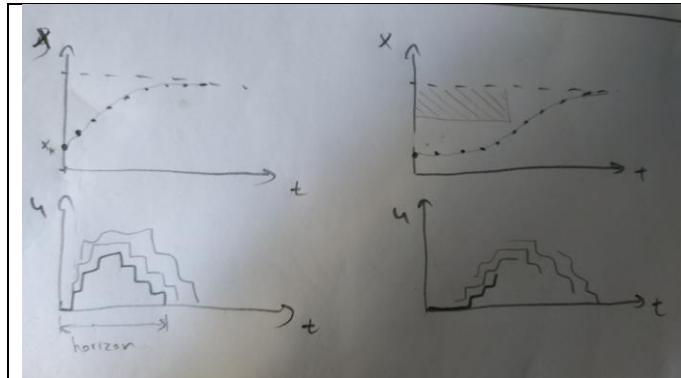
- Half the time, set $z=0$
- Predict next frame
- backprop



meaning of the condition x . The VAE doesn't simply has to reconstruct its input y , but it has to do it, taking into account that x happened. The representation of X is fed both to the encoder and to the decoder. It affects both. Notice also that we have a latent variable which represents the uncertainty of the environment, and we have to run this system with multiple values for z , in order to take multiple predictions (and corresponding action plans). Otherwise, we would have a blurry prediction.

The time over which we unfold, is called the horizon in MPC .

MPC does an optimization over the time horizon on the fly. So it is resource intensive.



an intuitive example of why mpc is better than just predicting the next immediate action only. Your target is to move the car in the target x . the cost is your distance from that position. Imagine that there is an obstacle. If you only minimize the cost of one action you might just move vertically. But if you minimize the cost over a long enough horizon then your next action could be just continue moving forward and turn a few steps afterwards.

Transformers?

To do

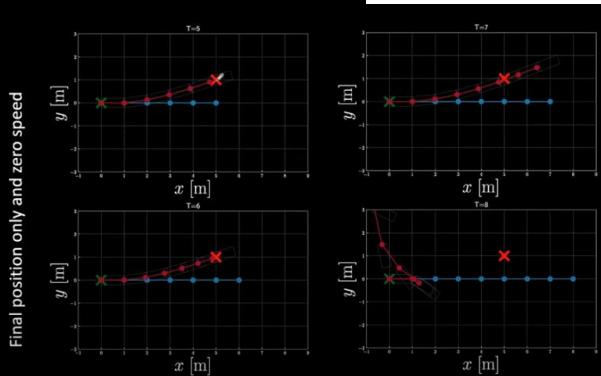
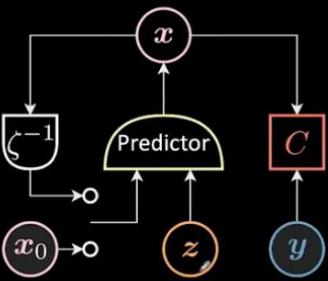
Alfredo's labs

Action plan

- Model predictive control
 - Backprop through kinematic equation
 - Minimisation wrt the latent
- Truck backer-upper
 - Learning an emulator of the kinematics from observations
 - Training a policy (this ~~no one made it work~~)
- PPUU
 - Stochastic environment
 - Uncertainty minimisation
 - Latent decoupling

He didn't make the notebook work

Model predictive control (MPC)



In classical MPC the world model is known. It is a known function of the attributes of the agent (position and speed for example) and the control variables (acceleration and steering angle for example). in the diagram the predictor module is the model of the world that predicts the next state from the previous one x_0 and the control parameters z . the state of the world in this example is a 1d vector of 4 attributes [x-position, y-position, angle, speed]. the y vector is the target state, with a target position and speed. in more complex systems, we don't know a function for the world, and we must learn one with DL. (This is model based RL)

In MPC you unroll this RNN like structure in time for a short time period (for example 5 steps) and you run an optimization problem with gradient descent for example, where you backpropagate through time w.r.t the control parameters. You find a set of actions for the next 5 timesteps. You take the first timestep's action. Then you repeat the optimization problem with a new window forward.

In the diagram, blue is the number of steps, red is the orbit of the agent, red cross is the target position. In the last example, it run for too many steps the agent passed the target tried to turn and it went further away, the algorithm diverged.

You can have MPC even if the world model is not known but learned with DL. You apply the same technique of unfolding in time, measuring the cost and minimizing it w.r.t to the parameters of the world model and the actions.

The truck backer upper case (Nguyen and Widrow, 1990)

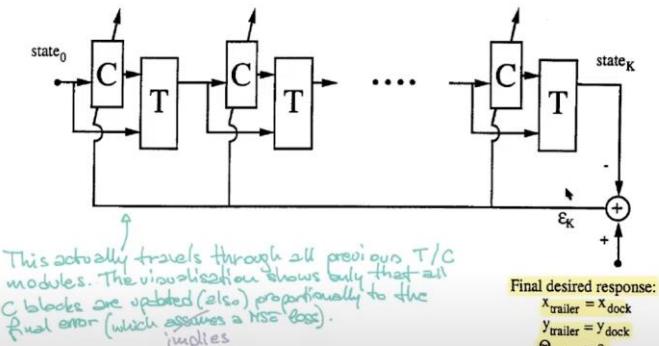


Figure 5: Training the controller with back-propagation

The T module is the one of the picture below.

The truck has two parts the front part and the back part. The target is for the back part to reach to a specific target vertically to the wall.

Before training the controller you have trained the truck dynamics emulator. Here you don't need a trained controller. You can just run many random steps to create training samples. Then you split that to training and test, and you run a training with these random moves. The target is for the emulator to learn to output the same state with the truck dynamics given a previous state and the control signal. it isn't necessary for the truck to be moving correctly towards the target. Any random move is adequate.

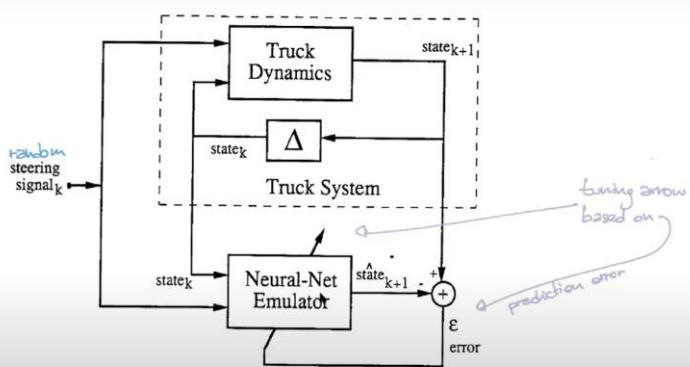


Figure 3: Training the neural-net truck emulator

To train the controller you unfold the network in time for k steps. K final is the timestep in which either of some conditions is being satisfied (φορτηγό διπλώνει, χτυπάει τοίχο, μέγιστος αριθμός iterations, target reached). Then at this final state the error is measured (MSE loss) and is back propagated through time w.r.t the parameters of the controller. The gradients are accumulated through time. The control is only one value, the steering angle. The speed is constant. So, you begin in a random position with random control. You make k steps till the end. This is one training sample let's for the controller. You measure the error and backpropagate it. You repeat with another random initial state. You do this for many iterations. This is how you train the controller.

Notice that this way the truck is trained for a specific target. If you change the target point you need new training. The truck will go to the previous target initially and will learn to move to the new target with a new training.

A note on how the intuition about the backprop. You get an error (MSE loss, squared distance). Its gradient is propagated first to the control module of the last timestep k. It will be a signal that the wheels should turn towards the target point. This is the gradient of the error w.r.t the control parameter. This is back propagated further. This last steering is affected by the previous steering. This will show you how the previous steering should be modified. And so on. Then we add all those gradients together.

In a policy network (a learned controller), an action is a nonlinear function of the state (because it is produced by a neural network)

An idea for a simple project: Add the position of the target point to the state as an extra attribute. Then train the model for various targets. The goal is to learn a controller that is independent of the target point, that it can reach any point. It learns to control based on the state which contains the target point. The cost is a function of the state in this case.

Handling uncertainty (in planning too)

Why to care about uncertainty	What is the level of uncertainty of a control action?
<p>• Cat / dog classifier, classify an hippopotamus</p> <p>• Reliability in steering control</p> <p>• Physics simulator prediction</p> <p>• Minimising action randomness when connected to a reward</p>	

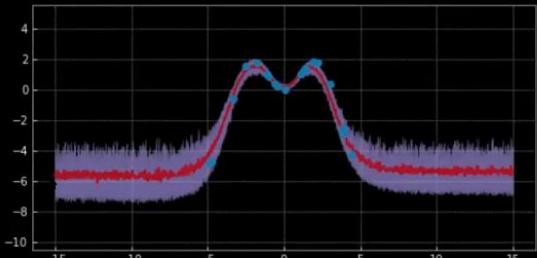
Bayesian networks (2016)

A Bayesian network (BN) is a probabilistic graphical model for representing knowledge about an uncertain domain where each node corresponds to a random variable and each edge represents the conditional probability for the corresponding random variables [9]. BNs are also called belief networks or Bayes nets.

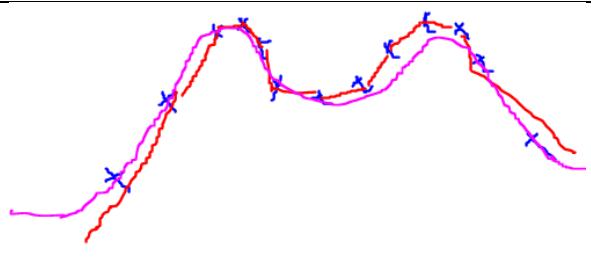
They are related with dropout. How? A way to measure uncertainty would be to train many models on the same data and measure the variance of their predictions. The larger the variance the larger the uncertainty on that region. Instead of training many different models and getting their predictions, you can train one model with dropout, and get predictions for many variations of it (dropout is random). In train mode (so that dropout is active) you do a loop of 100 iterations and you pass the datapoints to the trained network. So you get something like this.

Regression (I)

- See notebook demo



Red is the mean of all variations of the model. Purple is the variance of their predictions.



A side note for regression

If you use Relu, you get the red. It is a function made linear piecewise parts.

If you use sigmoid, it is a function made of non linear parts, it is smooth. It also becomes parallel in the edges.

PPUU Prediction and planning under uncertainty example (2018, NYU)

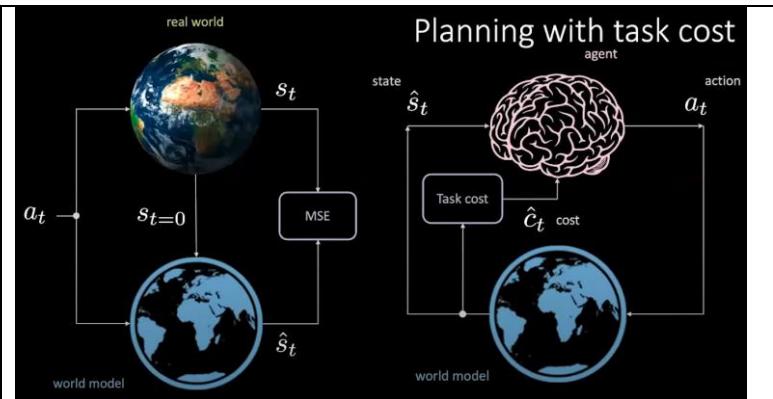
We must learn two things.

1. Learn a model of the world that handles uncertainty (predict the next video frames, which contain the positions of the cars). There were challenges, for example action insensitivity. It took them 8 months to do it correctly
2. Learn a policy. Learn a controller that can produce actions which minimize a cost (where the cost is part of the world, the cost is a (differentiable) function of the state). It took them 6 months to do it. 2 people were involved in this project.

Notice the difference with a policy learned with RL in a simulated environment. In this case we try to learn a policy only from observations. We don't have the ability to try other actions and see what happens to the world. We don't have a simulation environment. We only have observed data and try to learn a policy from them.

Info

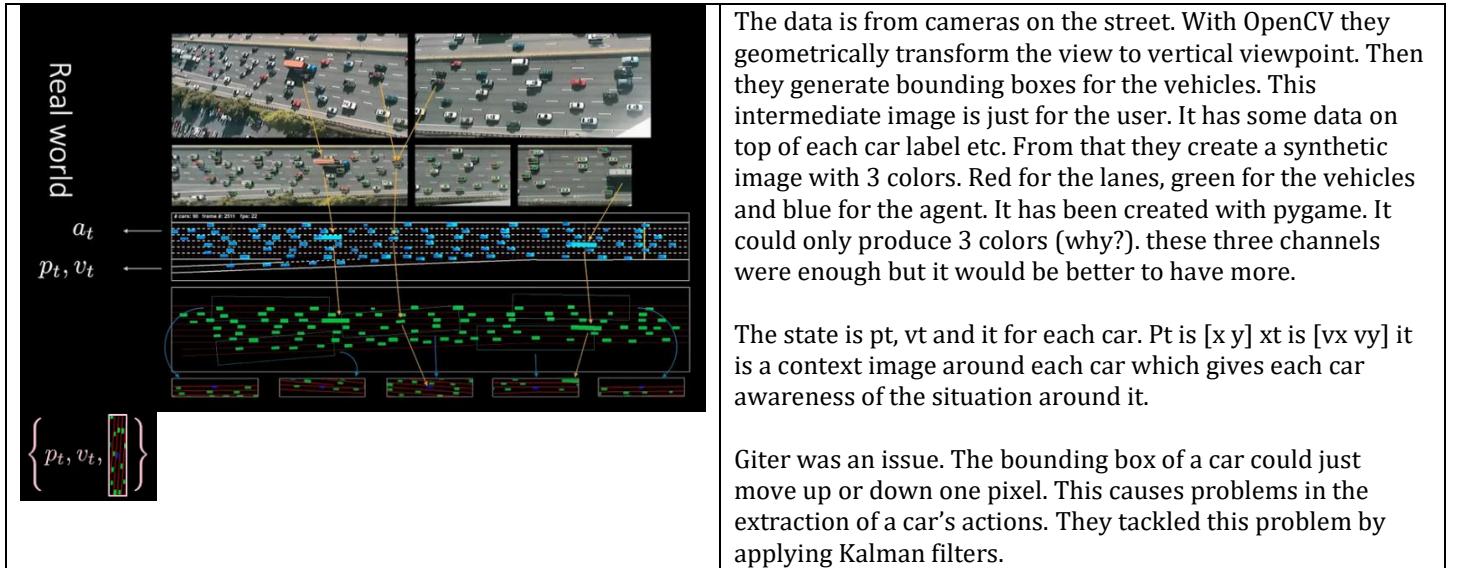
- Title: Prediction and Policy-learning Under Uncertainty (PPUU)
- Speaker: Alfredo Canziani [@alfcnz](#)
- Collaborators:
 - Mikael Henaff [@HenaffMikael](#)
 - Yann LeCun [@ylecun](#)
- Slides: bit.ly/PPUU-slides
- Article: bit.ly/PPUU-article
- Code: available in [PyTorch](#) on bit.ly/PPUU-code
- Website: bit.ly/PPUU-web
- Poster: bit.ly/PPUU-poster



1. Learning the world model

The goal of the first part is to learn a model of the world. This means to learn a model that can predict what happens next. It can predict the next video frames. The model doesn't just receive the state of the world and tries to predict the next state. We want to be able to predict the next states, taking into consideration the actions of an agent too. the agent is one of the vehicles of the video. Notice that we can extract their actions from the video using the tricycle kinematic model. By inverting it we can

extract the acceleration and steering angle from the change in the car's state. So, we have the car's actions and we can pass them to the model. Now, for each timestep we have a state and an action and the model has to predict the next state based on those.

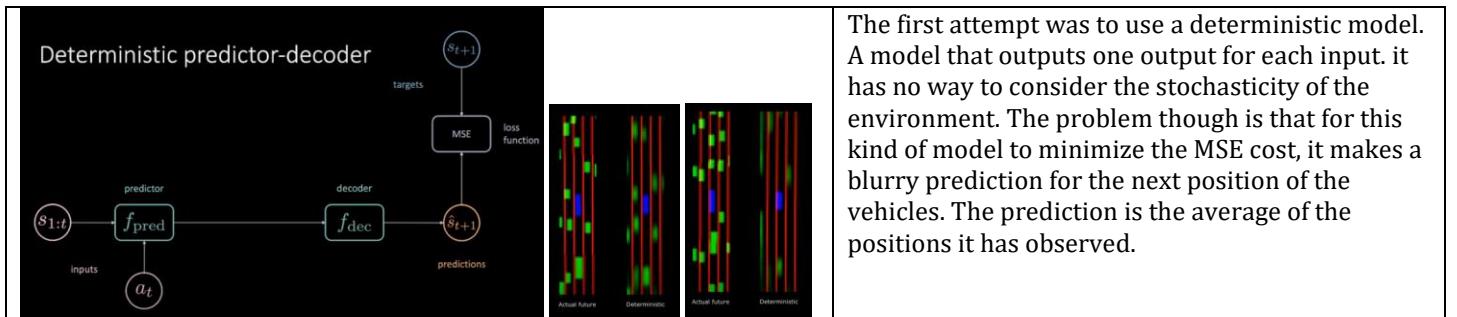


The model that process the state has the following general architecture. it has a CNN that is fed with the RGB context picture i_t and an MLP that is fed with a 4 components vector containing the x, y, vx, vy . It outputs 16 values for the mean and 16 for the std. (this was the architecture of the encoder of the VAE that creates μ and std but I guess it is the same with the predictor).

It is an autoregressive type of model, where the prediction is fed as input and you have a new prediction. So you can run this for as long as you want and see what the model predicts for long time periods.

Notice that the input to the model is not just the state of one timestep but for multiple ones, 1 through t ($1:t$). this is a common approach. In this case because we know the position and the speed of each car, we could just use one timestep. But if we only knew the position then we would need at least two timesteps to be able to calculate the speed and at least three for the acceleration. The common practice is to use a few timesteps. This way you give the model a history of the state.

Deterministic model



When you use an MSE for something that has multiple plausible outcomes, the model will learn to predict the average of them. This is what minimizes the cost. The average of some images is a blurred image.

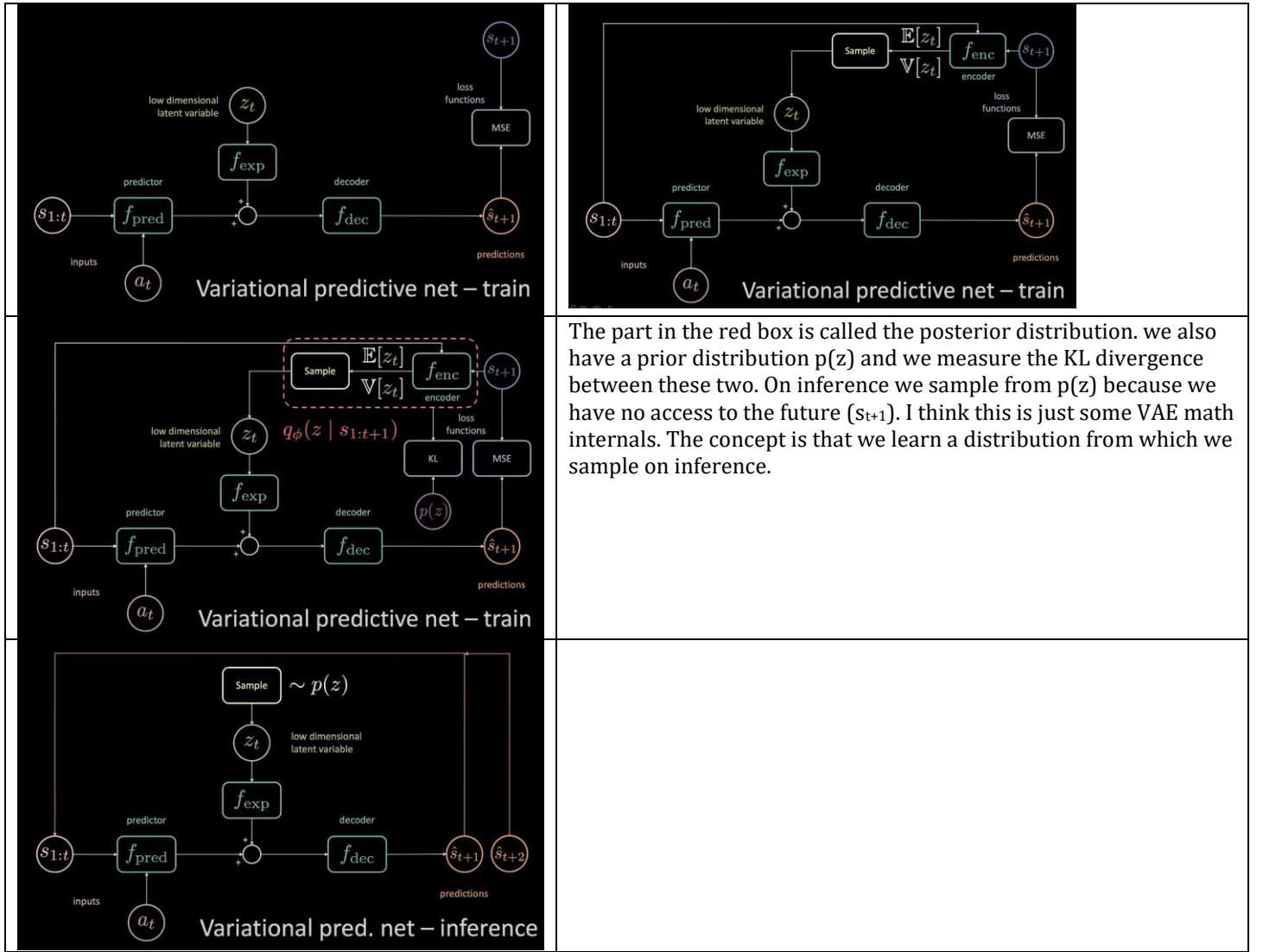
The MSE cost is between the predictions of a batch as I understand. You could ignore the batches and think of it as the MSE error of the predictions for all examples. The data contain some stochasticity. All being the same, the same car could behave

differently. The data contains this information. So for the same conditions, a car could have different positions which are all plausible. The model that tries to minimize the MSE would predict the average image of all these conditions, a blurred image. This is the one that minimizes the MSE. What if we somehow added a constraint about the car's size? It would predict at the center of the average image probably.

The main issue here is that the future is multimodal. Given an initial condition, the data set contains multiple evolution of the future. For example, say you have a car next to you. Once it goes faster and once it goes slower than you. If you want to predict both behaviours at the same time, you'll need a car that goes both faster and slower, meaning the car will stretch / elongate and its intensity vanish. (Imagine the superposition of both behaviours.)

Stochastic model

How we design this z though? With a conditional VAE (the output of the f_{pred} processing the $s_{1:t}$ and the a_t is the condition x).

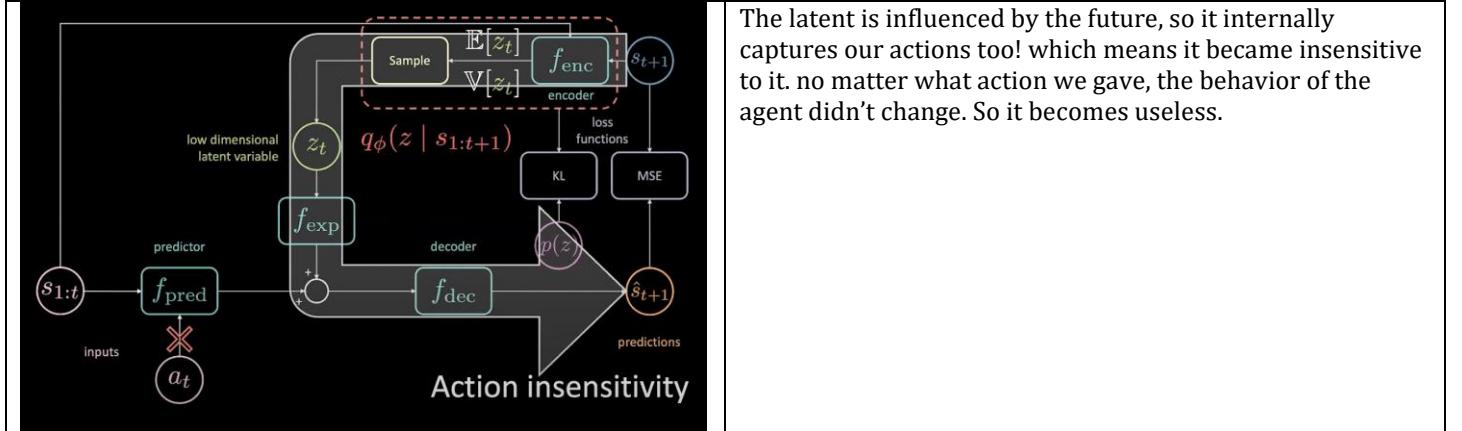


Transformers instead of CNNs

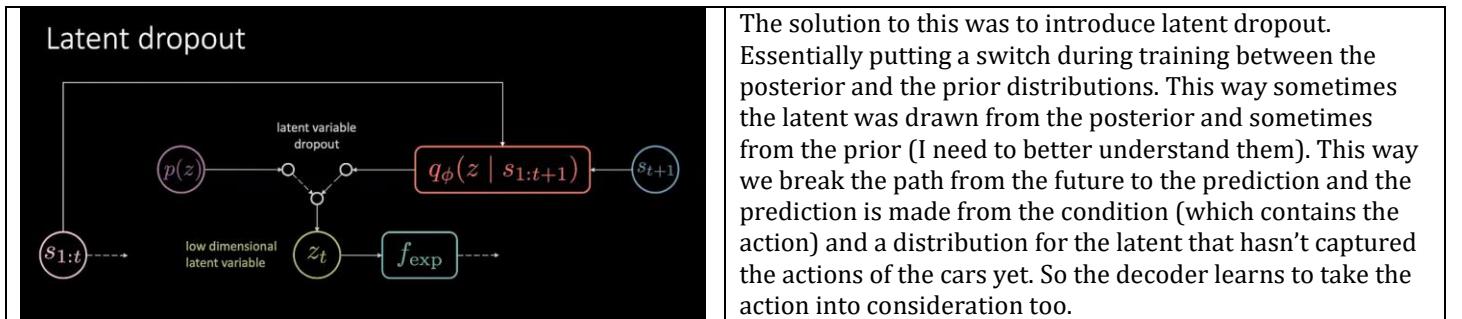
They used CNNs because they represented the positions of the cars as pixels (in the synthetic images). They didn't know how to work with sets of objects in 2018, since transformers that can do this, were very new then. Today they would probably used transformers. How exactly? What would be the difference? Transformers are permutation invariant (they work nicely with sets of objects) so they would be useful as I understood to predict the positions of the cars.

Important: Action insensitivity problem

The latent must encode what the other cars are doing. The issue is that during training, it learns to encode our agent's actions as well.



Why this happens? We train a conditional VAE to predict the future state (which means there is a path going from future to the prediction). In training our agent is just one of the cars of the video. We have just inferred its actions and passed them as input. the target is to learn to encode the randomness of the other cars behavior in the latent. But because our car is also part of the training data, and the latent is too powerful, it not only learns to encode the behavior of the other cars, but of our agent too. So it learns to encode the fact that if the lane turns, all the cars turn, including our agent. But our agent should depend on the action. So the information capacity of the latent is large and it isn't affected by the condition of our model. It is as if it is an unconditional VAE which sees the video and learns to reconstruct it. we want a conditional VAE instead. So we need to limit the information capacity of the latent variable.

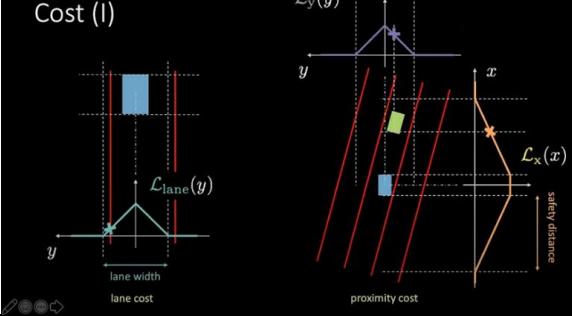
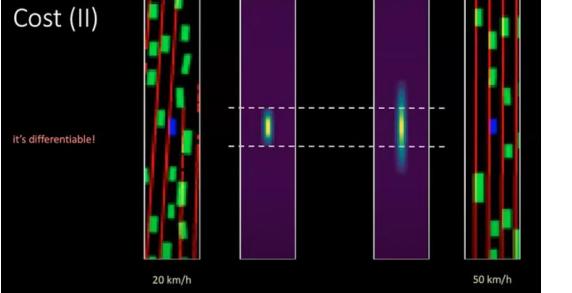


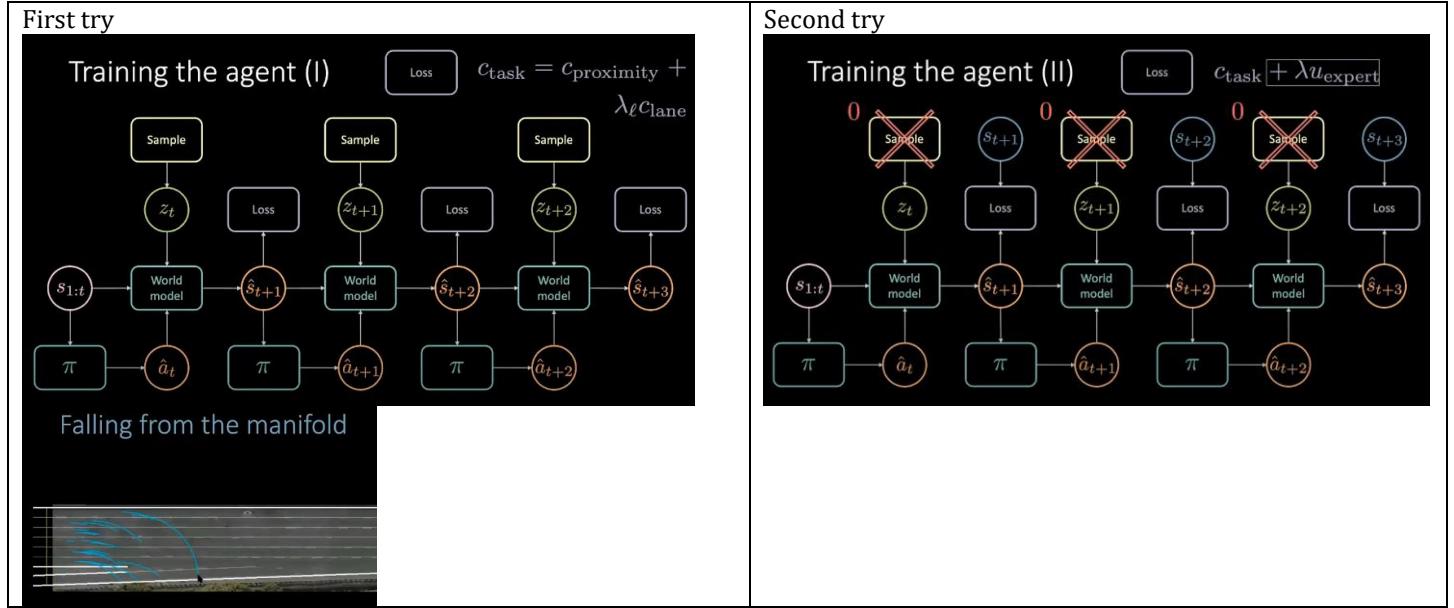
2. Learning a policy from observations

The first approach is identical with the model predictive control described previously. The problem is that it doesn't work. The policy goes to places where the prediction doesn't work (crashing to cars, falling outside of the lanes) where the prediction isn't trained on. So the world model produces a black picture and this picture has 0 cost. So the policy learns to go there.

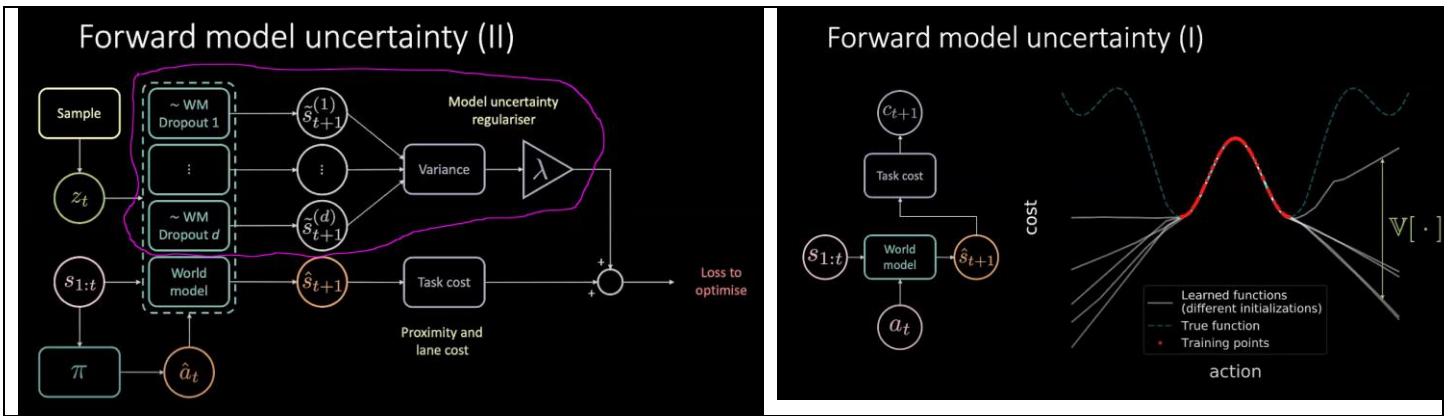
Second try: expert mimicking. The prediction should not go too far away from the actual positions after the action of the experts. But what actually happens is reflected by a specific latent value. Notice what the new problem is: the prediction closest to the expert will be the average image, the prediction of the deterministic model without the latent variable (we can go to the deterministic version by increasing the temperature). If you pick a latent, you will get a real solution but it will probably not be the same with the actual one and the error will be large. So you have to use the deterministic model. It works. It learns a policy. But this technique is kind of limiting. You are only trained on one possible future. Me: the deterministic model well for a short amount of time and I guess that a problem is that we can't make long action plans on it (if there is no other limiting factor).

Third try: minimize uncertainty (uncertainty regularization). And it works very well. We introduce dropout to the world model. So for the same state, action and z, we will get different outputs. We calculate the variance of these outputs and add that to the cost. So now we try to learn a policy, that generates actions which keep the car on lane and distance (the previous cost) but also minimize uncertainty of our prediction model. So it learns to generate actions that are within the knowledge of the prediction model (the world model).

<h3>Cost (I)</h3> 	<p>Lane and proximity cost The proximity cost has a component on x proximity and another for y proximity. X safety distance depends on speed.</p>
<h3>Cost (II)</h3>  <p>it's differentiable!</p> <p>20 km/h 50 km/h</p>	<p>The cost is applied like this. We create a mask with the current proximity where purple is 0. Then we multiply it by the context image. The result is the cost. If there are values within the proximity they are multiplied with non zero.</p>



Third try



PPUU vs RL

This is a PPUU technique. It is more data efficient in relation to RL since it doesn't interact with the environment at all.

The uncertainty regularization in the training of the agent is how you force the agent to take actions that maintain the states in a state space that has been observed by the world model? Whereas in RL, since you have a simulator, you don't need to do this because you can purposely explore the action space for "adversarial actions" during training. Does this make sense?

Yup. But someone would need to create a realistic simulator with smart agents controlled by...? So, it's better to use data from a real environment.

controlled by other PPUU agents!

But then all could simply go to the same speed and move uniformly. This would suggest that one should train a latent variable policy (whereas in this example we had latent variable world model) which encodes an individual driving behavior.

Very interesting project! For the policy training you use the unfolding in time technique (model predictive control). Can I ask, how many timesteps does the moving window has? I guess that it would be desirable to unfold for very large number of timesteps because this would mean that we can make really long action plans (long into the future). What is the limiting factor on the number of timesteps? Is it the vanishing gradients (like in a long RNN)?

Yes, of course, you need to start learning close to the target, then increase the distance.

JEPAs and Hierarchical JEPAs and the future

joint embedding predictive architectures

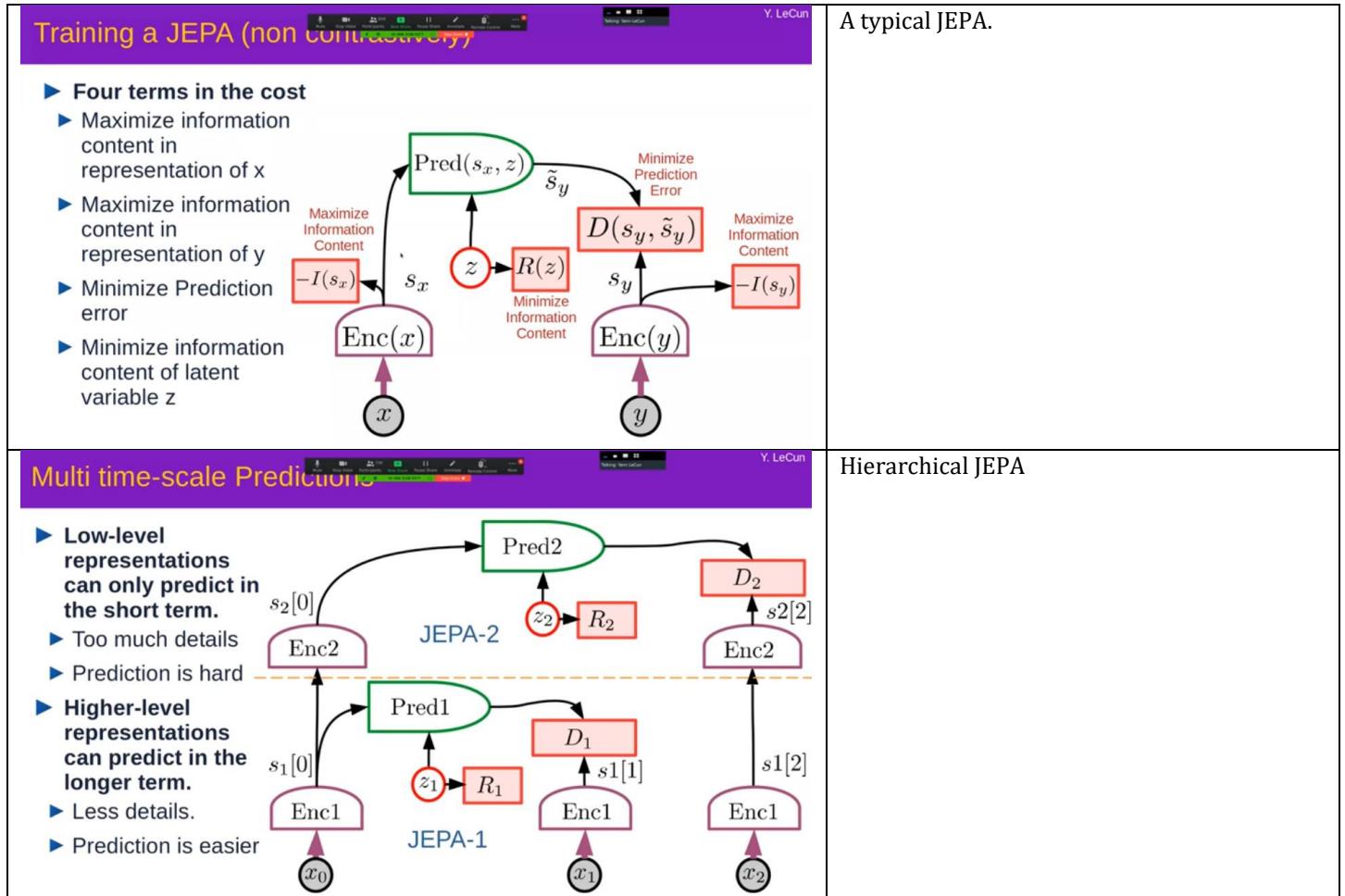
<https://youtu.be/DokLw1tILlw?t=1814> JEPAs explained and future plans

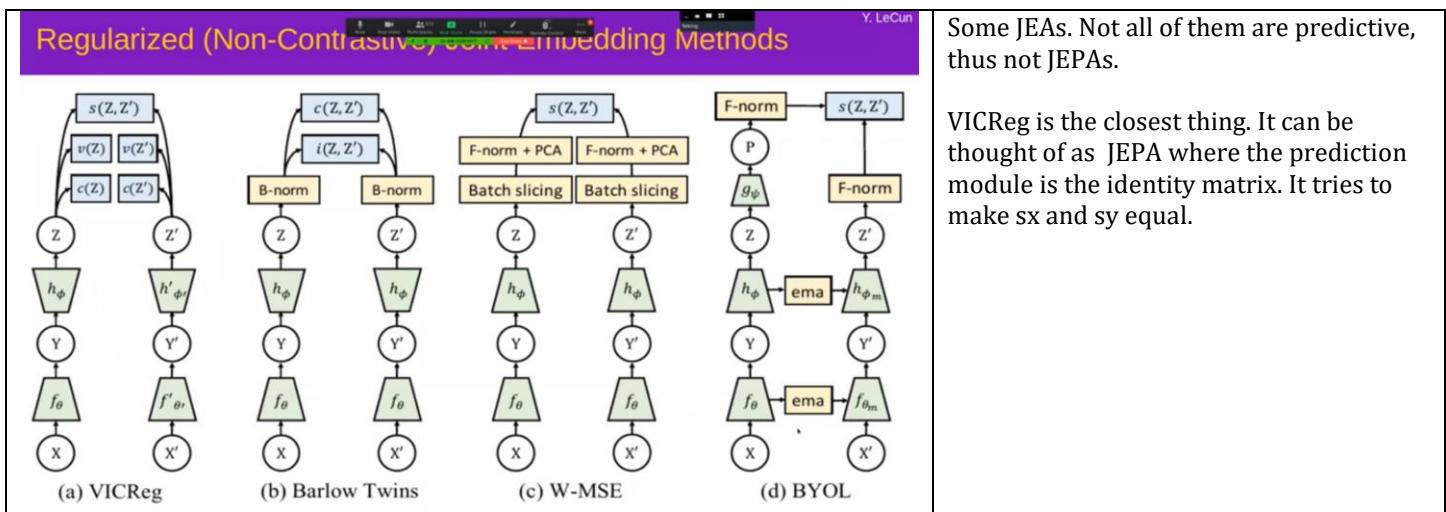
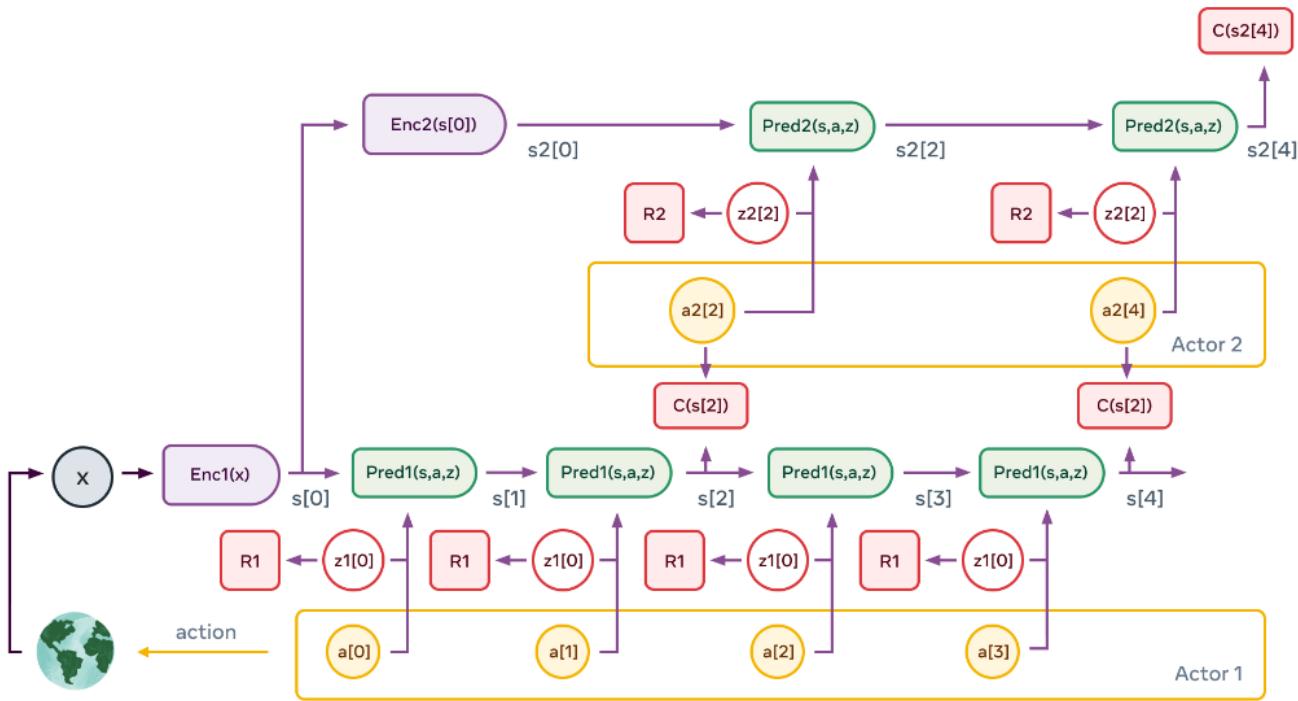
<https://ai.facebook.com/blog/yann-lecun-advances-in-ai-research/>

In the car example the model predicts in pixel space. It predicts what will be the next frame. It uses a generative model for this (a VAE). Lecun is now convinced that predicting in input space will not work for long term planning (he changed his mind on this, initially he believed it was necessary). JEPAs instead predict in the representation space. This makes a huge difference. Because you only have to predict the most important parts of y . there are 4 costs. First you want s_x and s_y to have as much information about x and y as possible. Then you want s_y to be easily predictable from s_x . This is what restricts the information stored in s_x and s_y . And if you have a latent variable to handle uncertainty, you have a regularization cost for it too. you can stack JEPAs on top of each other. The higher JEPAs work with higher representations and predicts further in time. It doesn't have to deal with details. With multiple layers of these, you start from the top to extract the long term goals, and you move

downwards from these, all the way down to millisecond by millisecond muscle control (which doesn't have to be instantiated from the start of planning, but they will be as the agent goes). This proposal hasn't been yet been implemented by lecun. This is a plan. This paper from Berceley 2022 though (Deep Hierarchical Planning from Pixels <https://arxiv.org/abs/2206.04114>) describes a model, Director that does "hierarchical behaviors directly from pixels by planning inside the latent space of a learned world model".

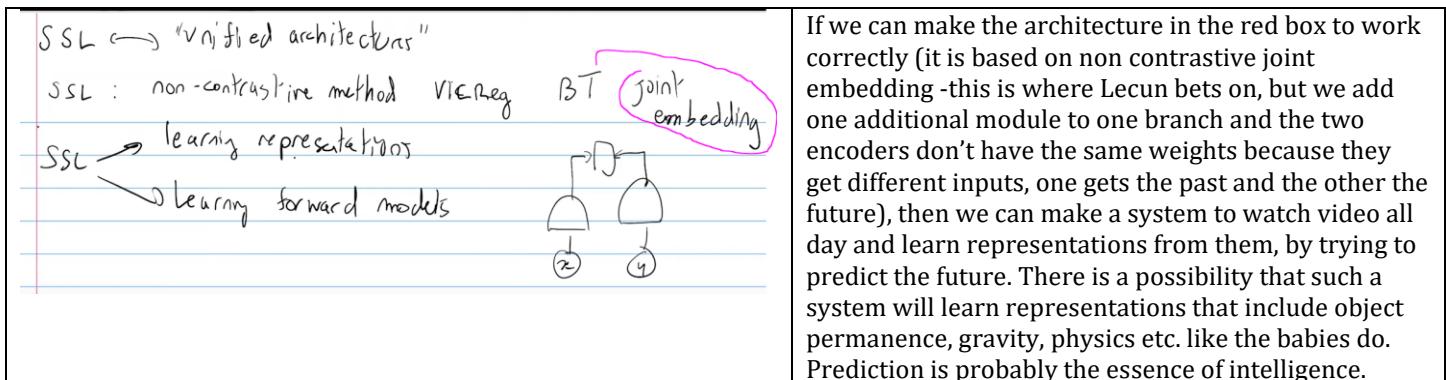
You can use these JEPAs as a basis for a forward model (a world model) for an autonomous intelligent system.

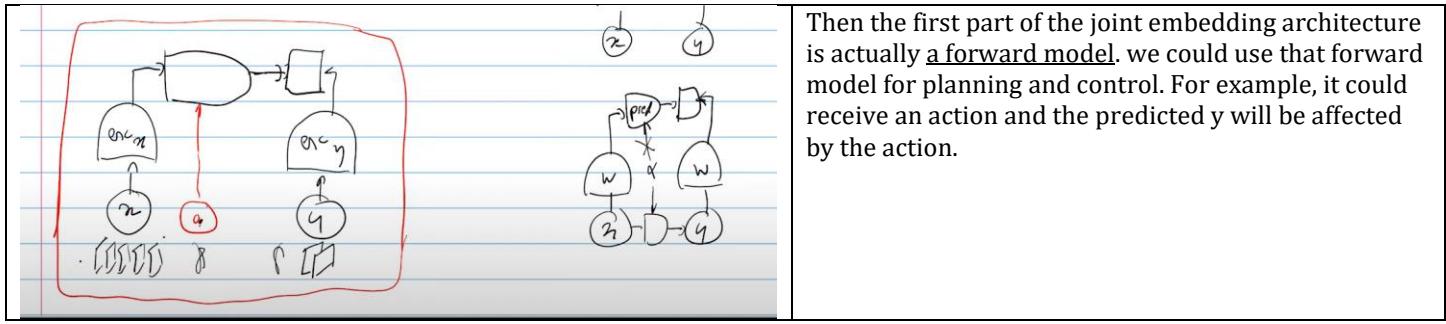




The future of AI from Lecun course 2021 (describing why JEPA without mentioning JEPA)

<https://youtu.be/MJfnamMFylo?t=2376>

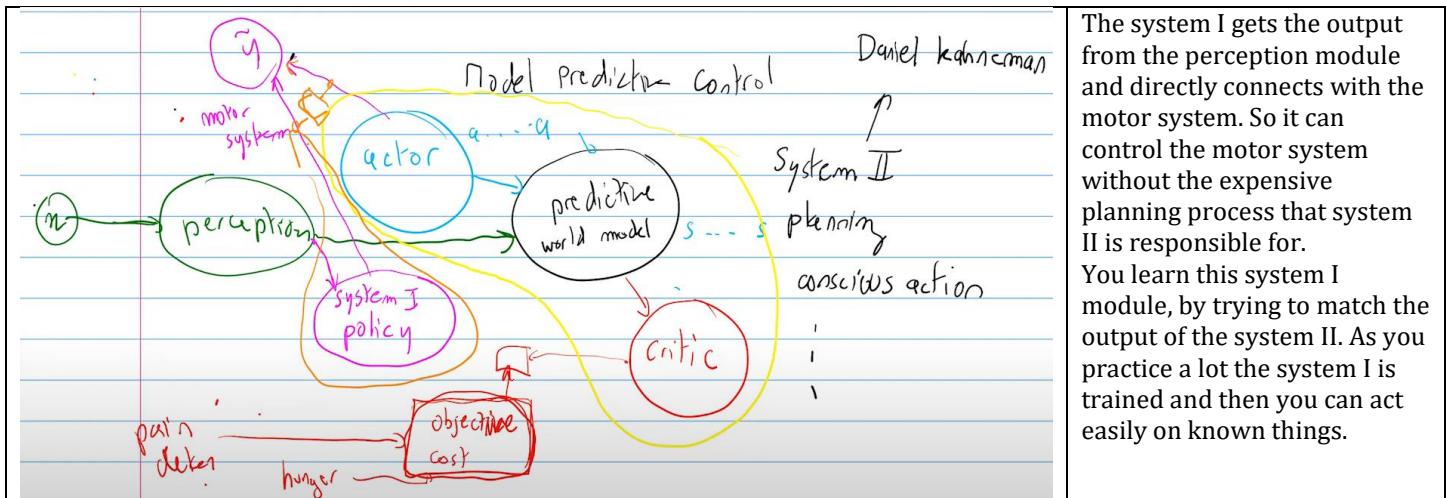




Then the first part of the joint embedding architecture is actually a forward model. we could use that forward model for planning and control. For example, it could receive an action and the predicted y will be affected by the action.

The important thing is that the two encoders are different. If we can manage to train systems like this without energy collapse, we might be able to create good world models just from watching video.

Autonomous intelligent systems



The system I gets the output from the perception module and directly connects with the motor system. So it can control the motor system without the expensive planning process that system II is responsible for. You learn this system I module, by trying to match the output of the system II. As you practice a lot the system I is trained and then you can act easily on known things.

The thing we can't do as of 2021, is learning good predictive world models that learn to predict under uncertainty. This is the main obstacle to create such systems.

This system contains model predictive control. MPC contains an optimization process to calculate the best actions for the current window (the horizon). Have in mind that Lecun mentioned amortized inference technique as a way to approximate the solution of an optimization problem (for finding the best z) which is much faster. Maybe it can be applied to this MPC. Or maybe it is a possible mechanism for system I?

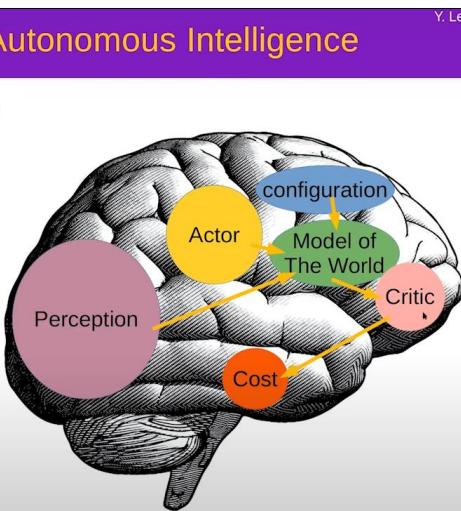
A possible architecture for an autonomous system proposed by Lecun. In our brains the cost is produced by the basal ganglia. This is that defines our satisfaction or dissatisfaction for everything.

Speculation: Architecture for Autonomous Intelligence

- ▶ **World Model:** predicts future states
- ▶ **Critic:** predicts expected objective
- ▶ **Cost:** computes objective
- ▶ **Perception:** estimates world state
- ▶ **Actor:** computes action

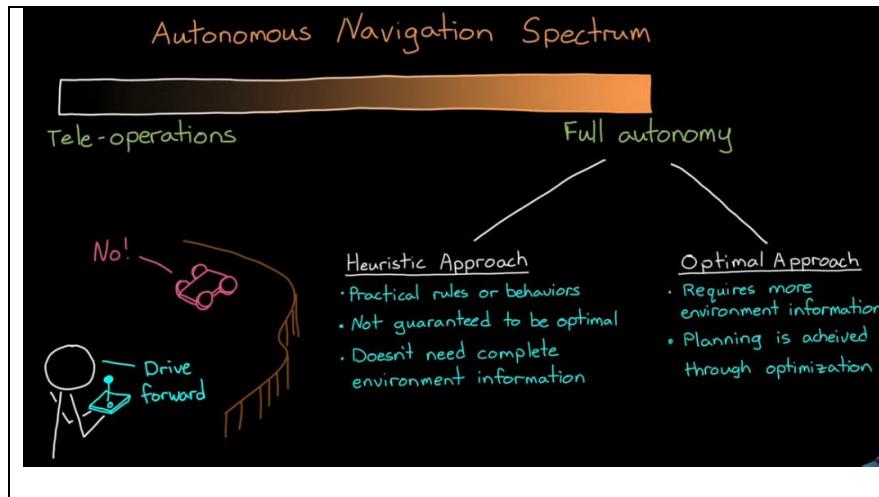
- ▶ Humans only have one world model engine!

- ▶ Configurator: configures the world model engine for the situation at hand.
- ▶ Is this what consciousness is?



We humans can only focus at one thing at a time. We can think of one thing at a time. Perhaps this is because we have only one world model engine, a system that represents a world model, and it is fine tuned (configured) each time for the task at hand. This is the role of the configurator model. A system that generates the settings which configure the world model.

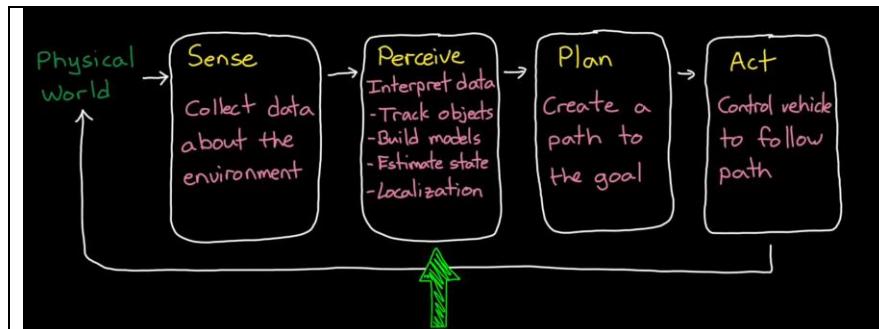
SLAM



You can have autonomous navigation without much environment information. The path will not be optimal though. For example a vacuum cleaner doesn't follow an optimal path to cover the map. it just uses a heuristic algorithm. it follows a path until it reaches an obstacle and then it turns randomly in another direction. After a while it will cover the whole map. not optimally though. Another heuristic algorithm might be go straight and when facing a wall turn left. Just this, could solve a simple maze if you are lucky enough. But in general you want optimal solutions that require a model of the environment (a map) and a motion planning algorithm.

In most cases optimal and heuristic methods complement one another. You might not be able to find the optimal solution because of incomplete knowledge of the environment. For example you might not see in front of your front car, so you can't plan an optimal path for passing it. in these cases you can implement a heuristic rule, something like always pass slower cars if it is safe. Then you can make an optimal path for switching lanes.

Simultaneous localization and mapping in a nutshell



Joint map and pose estimation. You have to create a map and localize yourself within that map at the same time. If localization estimate (pose) is given then mapping becomes easier. If mapping is given then localization becomes easier. But doing both simultaneously is more challenging.

Typically a SLAM system has two parts, frontend and backend. In the front end the Sensor data is transformed to an intermediary representation like optimization constraints, probability distribution for a landmark. The backend takes the

intermediary data and solves the state estimation problem (which is formulated as an optimization problem in pose graph approaches). There are two approaches for solving SLAM in the backend. Filtering methods and Smoothing methods. Filtering methods model the problem as an online state estimation. State is estimated on the go with the latest estimates. In smoothing methods (pose graph for example) full trajectories are estimated using the complete set of measurements, not only the latest.

Filtering methods

1. Extended Kalman filter (EKF)
2. Particle filter

Smoothing methods

3. Graph based SLAM approaches. Pose graph optimization (or more generically, factor graph optimization) is the de facto standard for most common SLAM implementations.

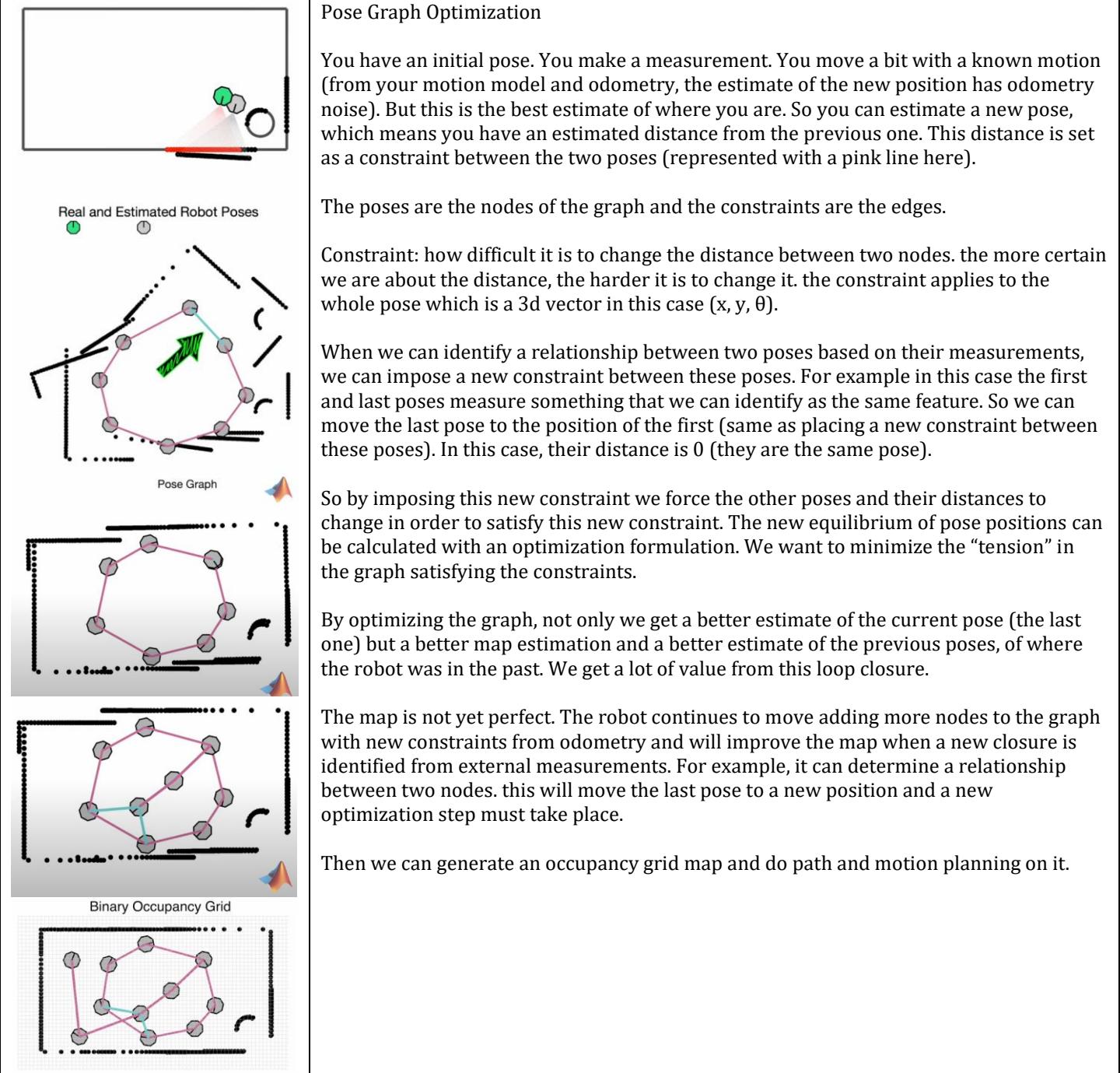
Dead reckoning

The process of calculating the current position from a past position given some relative measurements (like velocity, angular rate etc.). You use a model of the specific process, for example a car motion model. You have some relative measurements and you can calculate a position given a known model.

Odometry is a dead reckoning method. Odometry is the use of data from motion sensors to estimate change in position over time (wheel sensors for example counting number of wheel rotations). Measurement of agent's position and velocity. It estimates a process model, the model of agent's motion.

Optical odometry using optical flow. In robotics and computer vision, visual odometry is the process of determining the position and orientation of a robot by analyzing the associated camera images. Optical flow. Tracking features and their motion between camera frames. This is used for calculating an estimate of relative velocity of the sensor. You can then integrate it between frames to get the position. A form a dead reckoning.

Graph based SLAM approaches



Notice: You can't rely on relative measurements (IMU, wheel encoders etc.) to relate two nodes. you need external (absolute) measurements (lidar, camera, gps, etc.)

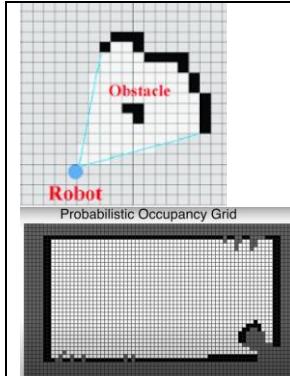
Notice: adding a wrong association (closure) will cause a lot of problems. So it is preferable to miss a correct association and add it later when you are more certain, than add a wrong one.

There are strong links between visual SLAM and bundle adjustment which is an older technique for photogrammetry. It is a special form of visual SLAM where you have a special constraint which is the reprojection error of pixel coordinates.

Path and motion planning

A Path is a sequence of poses between a starting and a goal pose. A pose in a 2d path finding problem is a 3d vector (x, y, θ orientation). Path planning is a subset of motion planning that is concerned with the path and its derivatives (velocity, acceleration etc.)

Occupancy grid map



One way to represent an environment is with a binary occupancy grid. It is a map usually created from sensor data. Space is split to grid cells of a certain area, and each grid cell has a value occupied or free, 1 or 0. (in a probabilistic grid, each cell has a probability for being occupied so it can take values between 0 and 1). We assume that each grid cell is independent from the others (for making math easier). You do navigation, path planning, localization in this map.

A typical algorithm used to estimate from sensor data, the probability for a grid cell to be occupied is called Static state binary Bayes filter (estimate probability distribution for every place).

Octree can be used for the occupancy grid map representation.

If you have an occupancy grid map you can start planning a path on this environment. In simple cases you can calculate a path analytically (for example straight line between two points in an empty map). but in most cases this is impossible and you use numerical methods to find the optimal solution.

Most common methods represent the grid cells of the map as nodes of a graph. Then we search the graph. There are methods that don't represent the map as a graph.

Using a graph to path finding, there are two approaches to creating that graph: Search based methods like A*, and sampling methods like RRT or RRT* (Rapidly Exploring Random Trees). Search methods, even efficient ones like A* that doesn't search the whole graph, are inefficient when the number of nodes (number of grid cells) is very large, 1000s or more. You can have many nodes in a 2d state problem (each state is x, y) if the number of grids is very large. You can have many nodes in high dimensional state problems (multi joint like robotic arms) where the space the grids have to cover is very high dimensional. In these cases we need to use sampling methods.

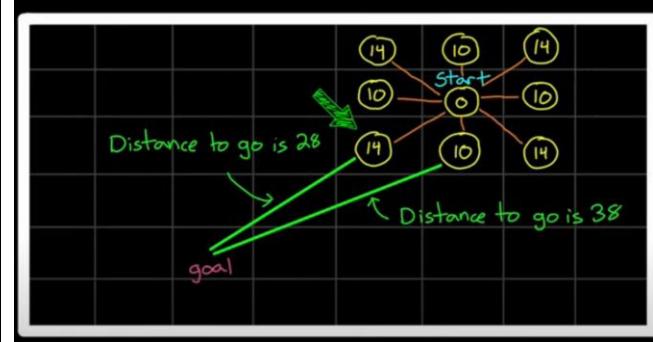
Graph based methods

Graph based methods discretize the environment into nodes. then we find a solution using only these nodes (not the intermediate node positions). The length (value) of the edges can represent distance, time, efficiency etc. some cost. Each node has also a value which is the total cost to reaching there from the starting node. Random wandering is the most naïve method for finding a path with a cost lower than a threshold. Notice that if we reach to the same node from a different path, we change its cost to the lowest of the two paths. Also we delete the edge that connects the two paths. This last action forces the graph to be a tree. A tree is what we want because ultimately we will have a branch for every node where the branch cost is the lowest possible for reaching at that node.

- **Search based planning**

Graph search algorithms: A* (1968), BFS, DFS, Dijkstra's algorithm.

We search the whole graph (all the cells) and assign a cost to each one (the cost at reaching to it). then we can select the optimal path (optimal at the resolution of the grid). But we must search the whole tree.



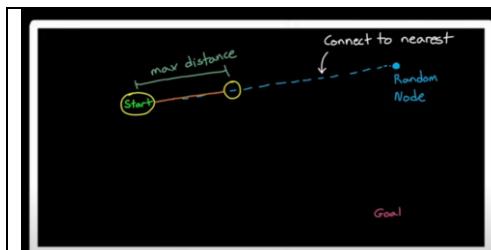
A* is an improvement that doesn't search the whole graph. It still adds nodes (searches) in an ordinary way (step by step, around the start) but it prioritizes nodes that have higher probability to be in the optimal path. It does so based on a given heuristic like the straight line distance of node to the goal node (in addition to the cost of the node). For example here the node with the cost 14 will be prioritized from the one with cost 10 and the algorithm will search (add nodes) around that node first. So we might end up to the goal with an optimal path without searching all nodes.

- Sampling based planning

PRM (probabilistic roadmap)

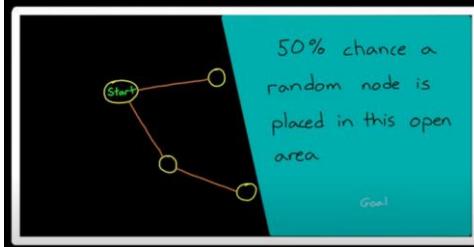
RRT and RRT* (Rapidly Exploring Random Trees)

RRT



Gives a valid path but not necessarily the optimal. As long as there is a path this method will find it for sure if samples reach to infinity.

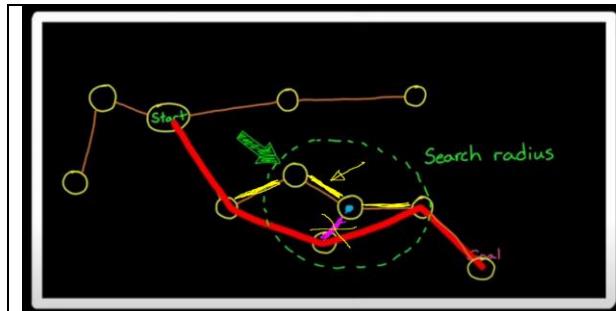
Instead of adding step by step the nodes as A*, we add random nodes at random distance from the start. We use a max distance parameter. If the distance is larger we put the node to max distance. If there is an obstacle between the start and the node, we don't add that node at all. we add the node by connecting it to the nearest neighbor.



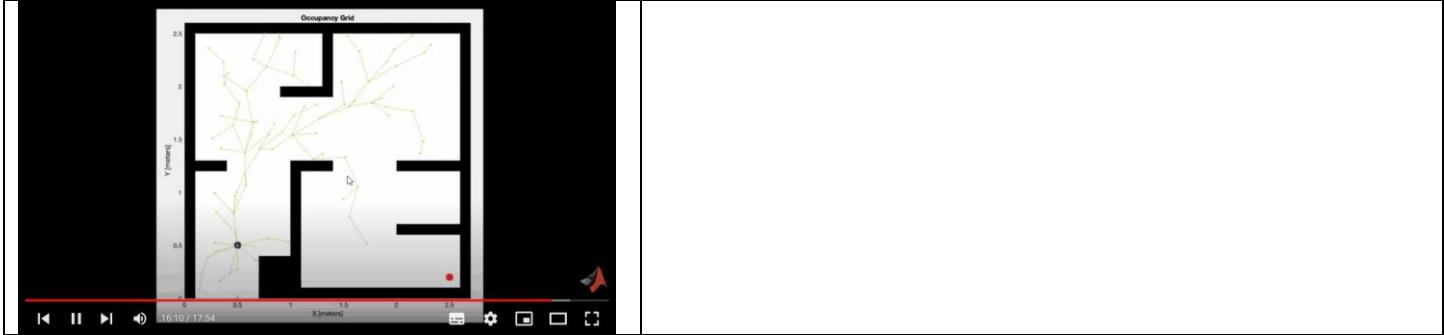
The greater the unexplored area, the more chances there are a new random node to end up there (because we pick randomly in the whole state space). Thus we rapidly search unexplored areas. As the unexplored area fills up, RRT fills the tree with more branches.

The fewer nodes added to the tree (fewer nodes searched) for finding the solution the faster the algorithm.

RRT*



We add nodes the same way as in RRT but instead of adding it to the nearest neighbor we search within a search radius around the node, and look if we can connect it with some other neighbor that will create a path with a smallest distance, while maintaining the tree structure.



Misc

From a post on tesla autopilot planning

Discrete search methods

Discrete search methods are really great at solving non-convex problems because they don't get stuck in local minima. But it does not perform well in solving high-dimensional problems, because it is discrete. It does not use any gradient information. So literally has to go and explore each point to know how good it is. This is very inefficient.

Continuous function optimization

Continuous function optimization is good at solving high-dimensional problems because they use gradient-based methods to very quickly go to a good solution. But for the non-convex problems, they can easily get stuck in local minima and produce poor solutions.

The Tesla AI team's solution is to break it down hierarchically. First, use a coarse search method to crunch down the non-convexity and come up with a convex corridor, and then use continuous optimization techniques to make the final smooth trajectory.

The high dimensionality comes because the car needs to plan for the next 10 to 15 seconds, and needs to produce the position, velocities, and acceleration of this entire window, this is a lot of parameters to produce at run time.

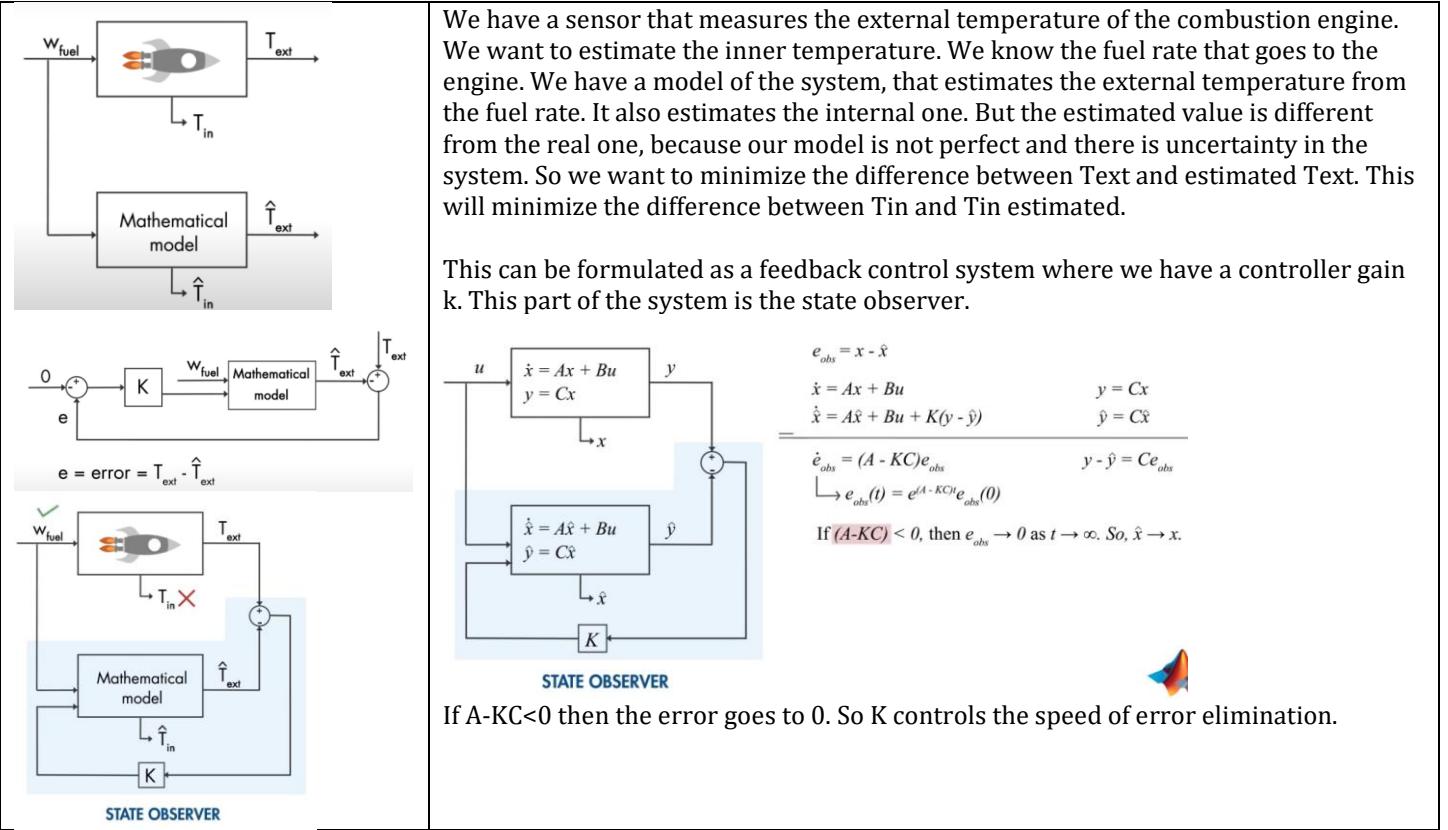
And the planner does thousands of such searches in a very short time span. 2500 searches are done in 1.5ms. This search speed is very fast. Imagine that if you use 60km/h, 1.5ms will only advance 2.5cm, and you already have 2500 driving routes. Finally, the planner chooses one based on the optimality conditions of safety, comfort, efficiency and easily makes the turn.

Fortunately, the parking problem in Tesla's Vector Space is like playing the Atari game of Ms. Pac-Man.

Filters

State observers

Kalman filter is an optimized state observer



In a control loop, the controller gain is the strength of action a controller will take at a particular point below or above the setpoint.

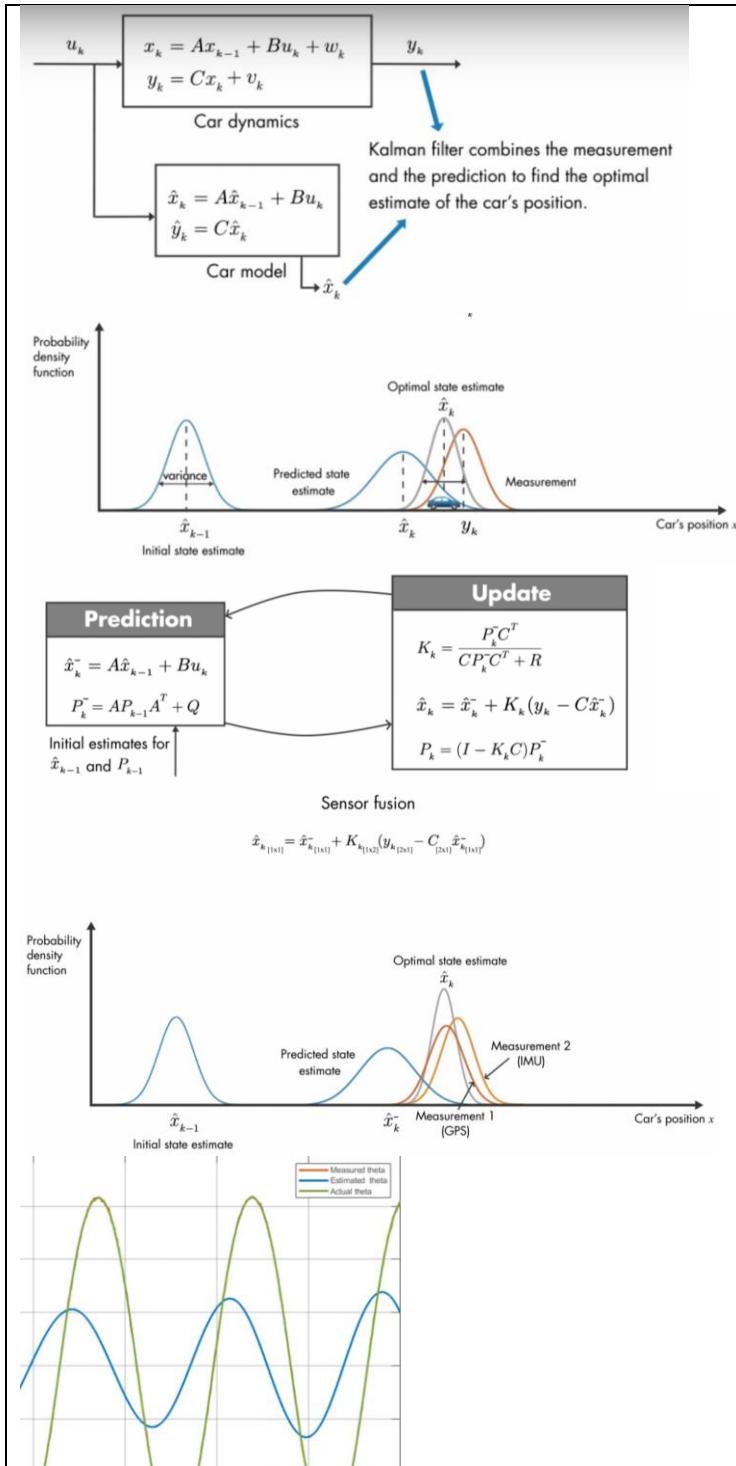
Kalman filter

An optimal estimation algorithm. the basic process is split in two parts, predict the state of the system through a model and then correct it with a measurement. It is used for:

1. Estimate the state of a system (the variables of interest) when they can't be measured directly. For example you can't place a sensor inside a combustion engine. You place it further away and you have to estimate the inner temperature.
2. Estimate the state of a system by combining noisy signals

We need to have a process model of the system (to estimate a state) and at least one measurement signal. From these we make an optimal estimation of the state and its covariance.

You have a car's velocity and you want to optimally estimate its location. You have a sensor that measures the location. You also have a process model that estimates the location. Notice that there is uncertainty both in the model's prediction and in the sensor's measurement. In the following diagram blue lines represent the car's location estimation predicted by the model of the system. The brown line is from the measurement. At time $k-1$ the model predicts a location $x(k-1)$. Due to uncertainty this is the mean of a gaussian distribution. the car could be anywhere within that distribution with the relative probability. At time step k the model will make a new prediction which would be even more uncertain because of accumulated uncertainty in the time period (road anomalies, tire condition etc.). at time k , we also make a measurement with the sensor (y_k is the sensor measurement value). It has also some uncertainty represented with a gaussian distribution. the real position of the car is somewhere along these two distributions. But what is the best estimate of this current position? It turns out that the best estimation is given by multiplying the two distributions. The resulting one has smaller variance from both. This is what the Kalman filter does. The state estimation from the process model is called the a-priori estimation. This state estimation is updated with the measurement and we get the a-posterior state estimation. If we minimize the variance of the a-posterior state's distribution we get the optimal state estimation.

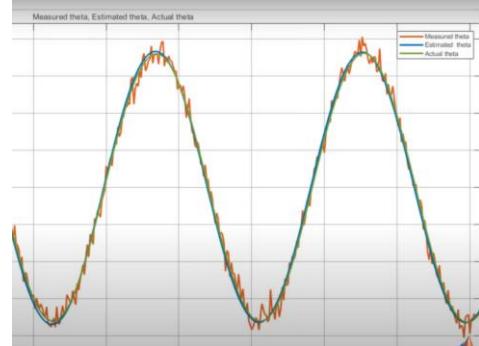


The goal of the KF is to minimize the variance of the optimal state prediction, by modifying k (the gain of the closed feedback system). The gain controls what percentage of the optimal estimation is given to the model prediction and what to the measurement. If the noise of the sensor is small (the variance of the measurement is small) then KF gives more weight to the measurement and vice versa.

KF calculates the current state estimation from the previous one (plus the measurement). It doesn't need older information (It is a recursive algorithm). We can continue the loop for more timesteps though, using the current state estimation as the initial state and estimating the next one. And so on.

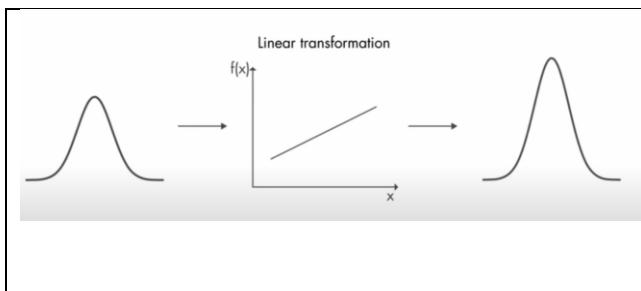
Notice that KF can use more than one measurement from different sensors (different distributions). It then multiplies three distributions instead of two. And in the formulas some variables become vectors. This is sensor fusion.

This is an example of a pendulum for small theta values (where the model is linear). The real angle theta, the measured theta and the estimated one with a Kalman filter. The KF filters out the noise and it is a very good estimate of the real theta.

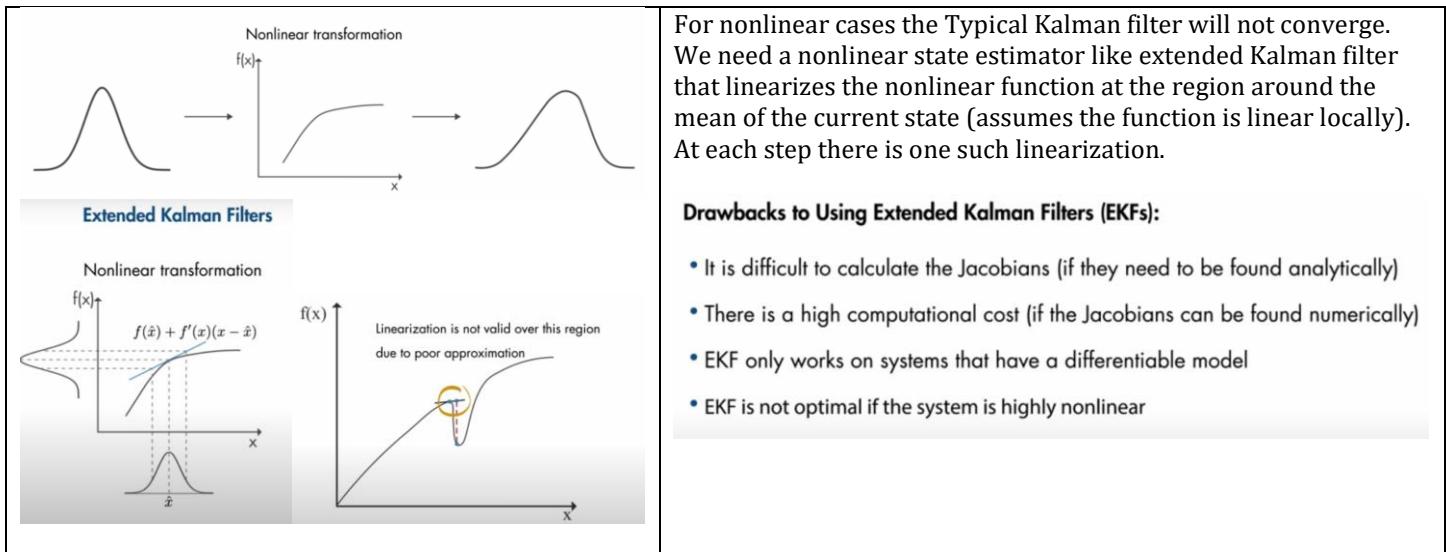


If we use large theta values though (large initial theta) the process model becomes non linear and KF doesn't estimate properly.

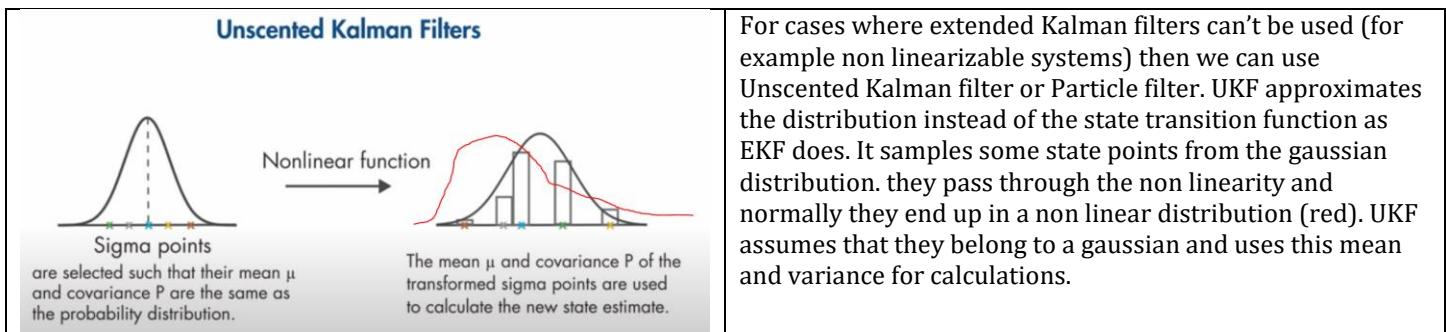
Extended Kalman filter



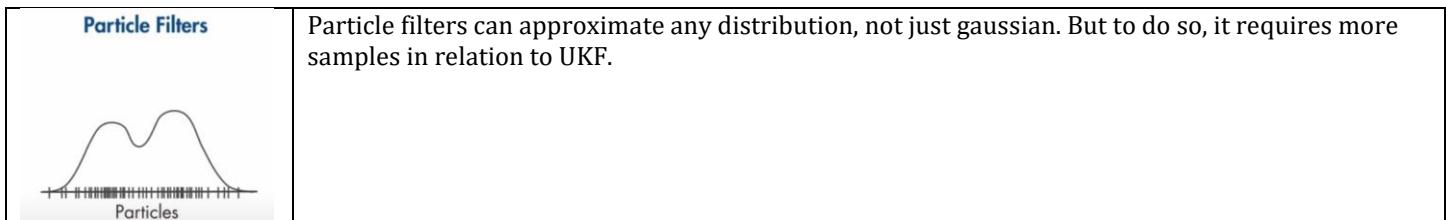
Notice that the Kalman filters are defined for gaussian distributions. This means linear systems, where the state transition function (here $x(k) = Ax(k-1)$) and the measurement function is linear. If a given state following a gaussian, passes through a linear transformation the result is still a gaussian. So if the state transition function is linear, then the predicted state will still follow a gaussian. The same is true for the measurement function.



Unscented Kalman filters

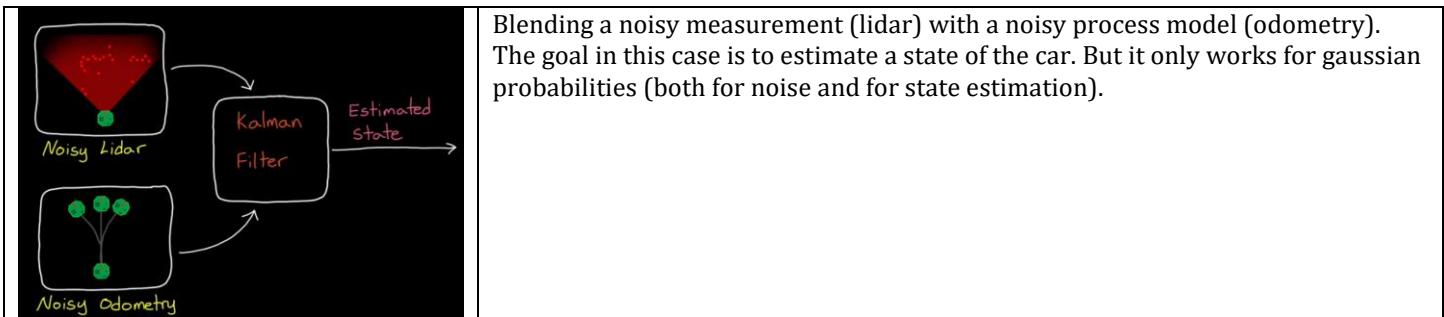


Particle filters

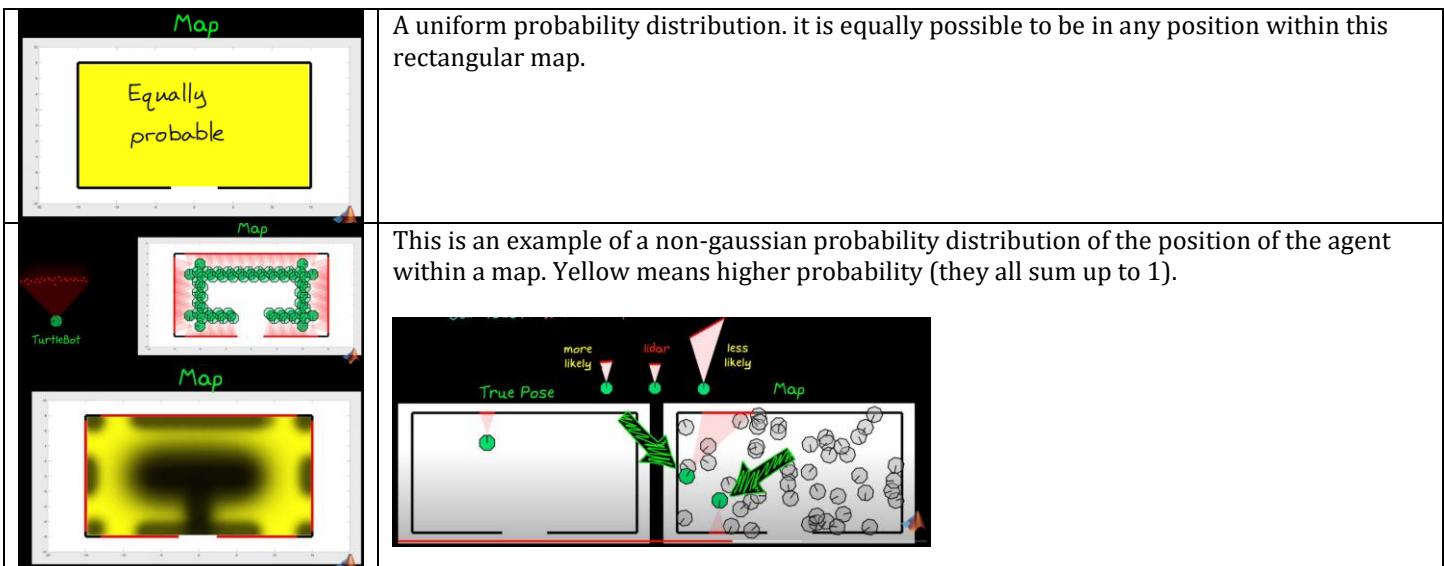


Filters comparison

State Estimator	Model	Assumed distribution	Computational cost	
Kalman filter (KF)	Linear	Gaussian	Low	
Extended Kalman filter (EKF)	Locally linear	Gaussian	Low (if the Jacobians need to be computed analytically) Medium (if the Jacobians can be computed numerically)	
Unscented Kalman filter (UKF)	Nonlinear	Gaussian	Medium	
Particle filter (PF)	Nonlinear	Non-Gaussian	High	



Particle filter



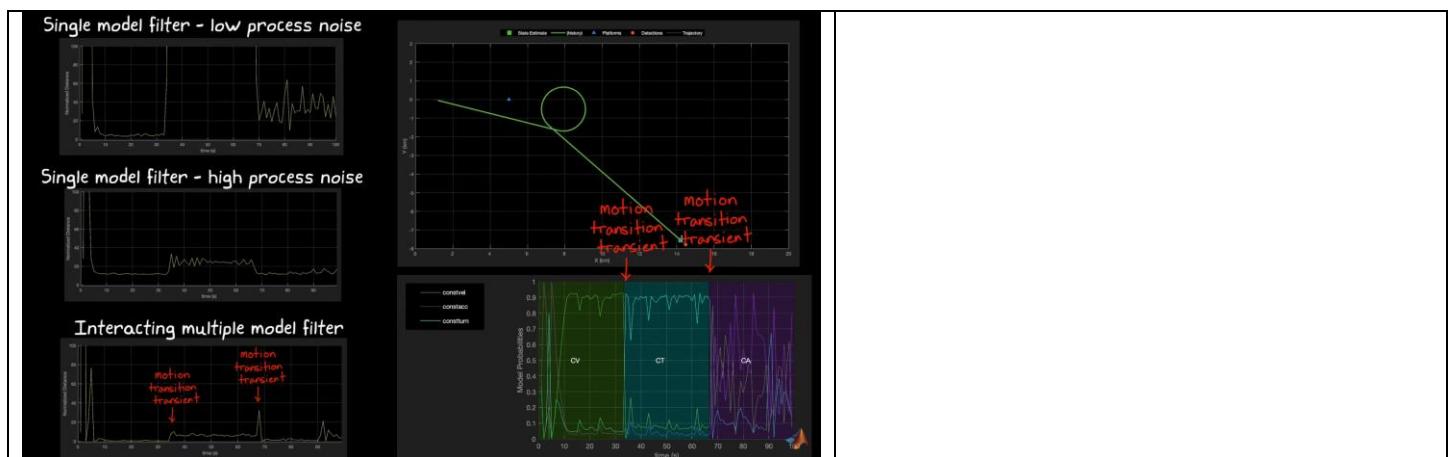
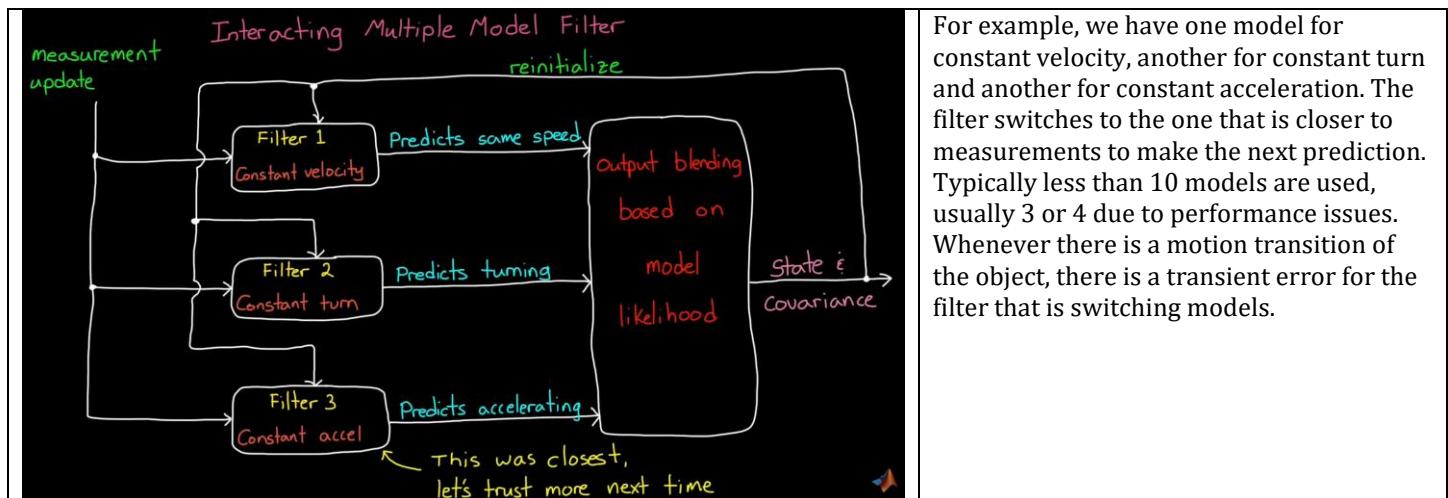
- You know the map and you want to find your position on the map.
- You begin with a uniform distribution. You can be anywhere on the map. You sample this distribution and you get let's say 100 particles. A particle is a candidate robot pose. Notice that the orientation is also randomly sampled.
- You make a measurement with your sensor (lidar in this case)
- What the lidar would have returned if the pose was the real one. Then compare with the actual measurement and determine how likely it is that this particle is the current state.
- So after one iteration the probability distribution is not uniform, but is skewed towards the most probable states. Then we are generating new particles by resampling from this new distribution. this random discrete resampling of the probability

distribution is why this method belongs to Monte Carlo localization approaches. As long as there are enough particles to represent the distribution the particle filter can handle a non-gaussian distribution. Tradeoff between accuracy and speed. more particles increases accuracy decreases efficiency.

- The car moves at a certain path and dead reckoning its relative position with onboard odometry.
- We apply this estimated motion to every particle in the filter (taking into account the noise of odometry)
- Then we measure again. And repeat the process. In new iteration we can start decreasing the number of particles (adaptive monte carlo localization AMCL)

Interacting multiple model filter (IMM)

Kalman filter is a single model estimation filter, which means that it uses only one model for predicting the state of the system. An IMM filter on the other hand uses multiple models of the system and at each time it uses the one that is closer to measurements. This filter is used for tracking other objects.

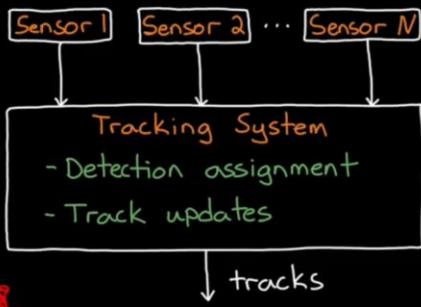


Object tracking

There are two different architectures for a tracker

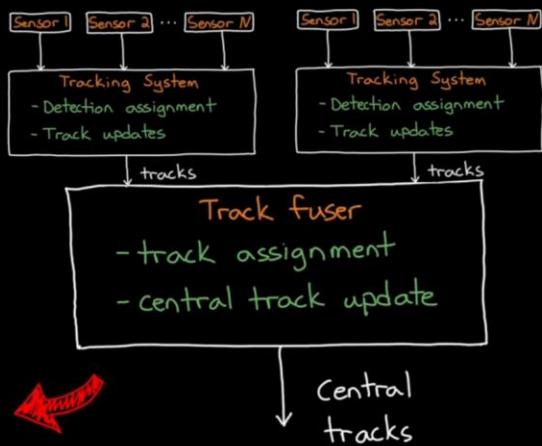
Two different tracking architectures

Central-level tracking



- * Single or multiple sensors
- * Single or multiple targets
- * Point or extended objects

Sensor-level tracking and track-level fusion



Track-level fusion benefits

- Bandwidth
- Compute
- Specialization
- Data access

Track-level fusion challenges

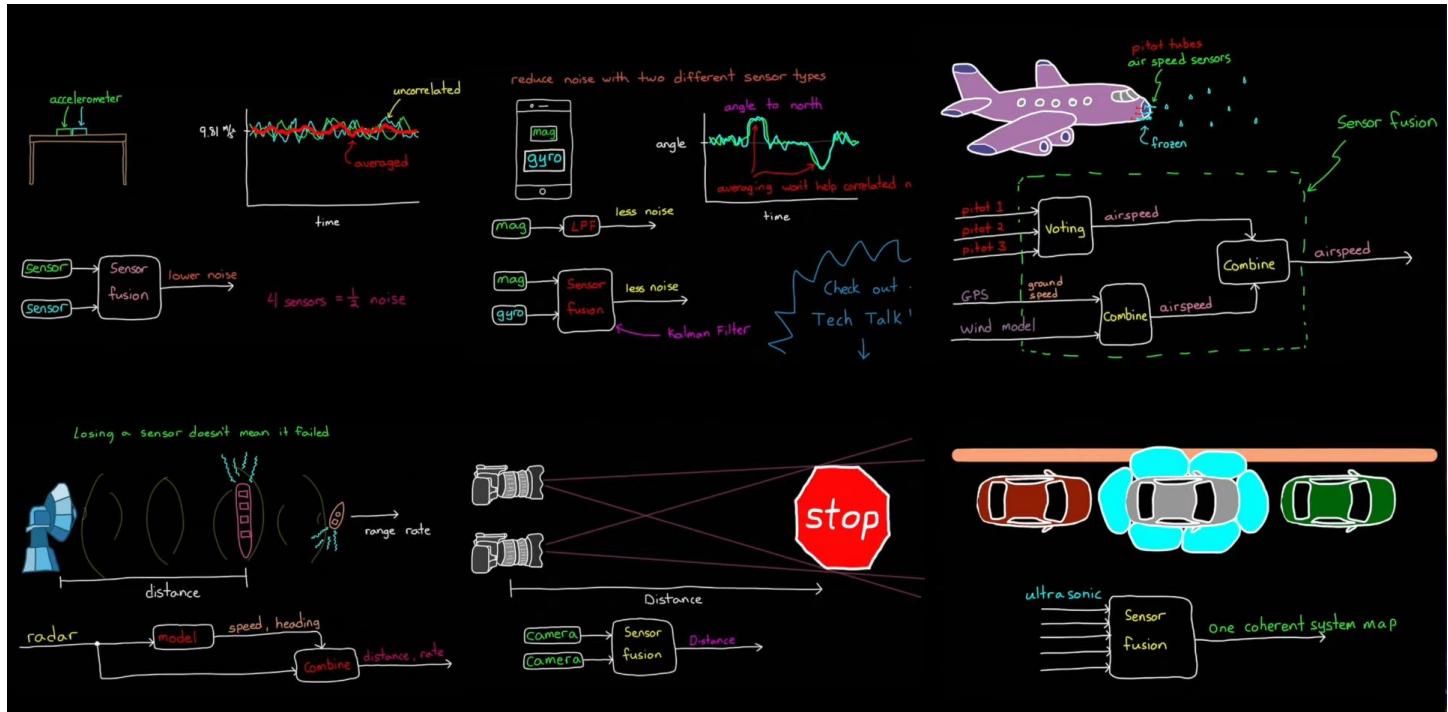
- Rumors
- Correlated noise
- Accuracy

$$\text{State} = \begin{bmatrix} \text{Position} \\ \text{Velocity} \\ \text{Orientation} \\ \text{Shape} \end{bmatrix} \quad \text{and covariances}$$

A track is a state vector and a set of covariance matrices which show how much confidence we have in this state.

Sensor fusion:

- reduces noise (cases 1, 2)
- increases reliability (cases 3, 4)
- estimates unmeasured states (case 5 depth estimation)
- increases coverage (case 6)

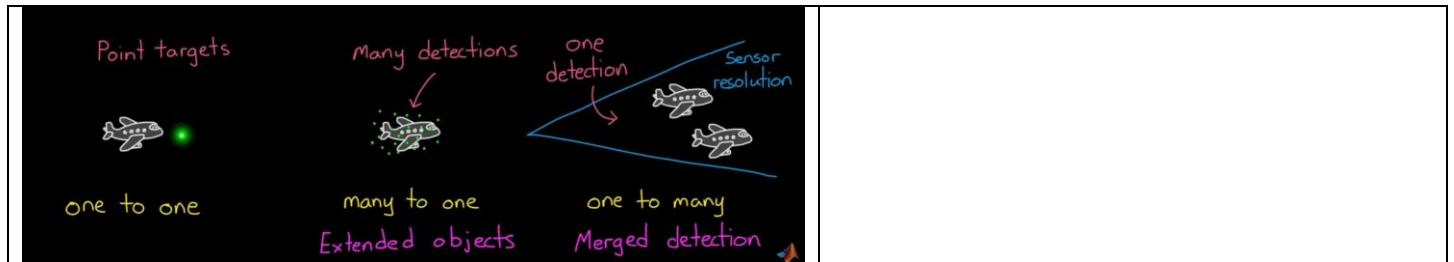


A tracker needs to

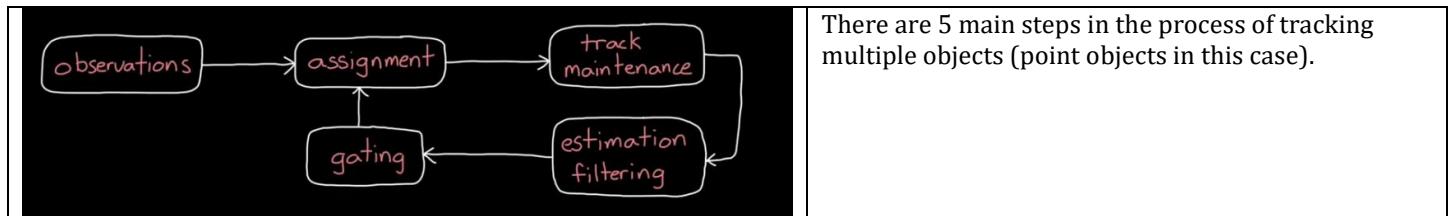
- Take in measurements
- Associate data to objects (noise and uncertainty makes that difficult)
- Maintain and delete tracks
- Determine motion models of the detected objects

It takes in measurements and updates the state and covariance of the tracked objects.

Point object trackers and Extended object tracking. In point object tracking we have one data point per object. Tracker has to estimate the number of objects in the scene, their position and kinematics). In Extended object tracking the object is larger than the resolution of the sensor so we have more than one data points for the same object.



➤ Multiple object tracking (point targets)



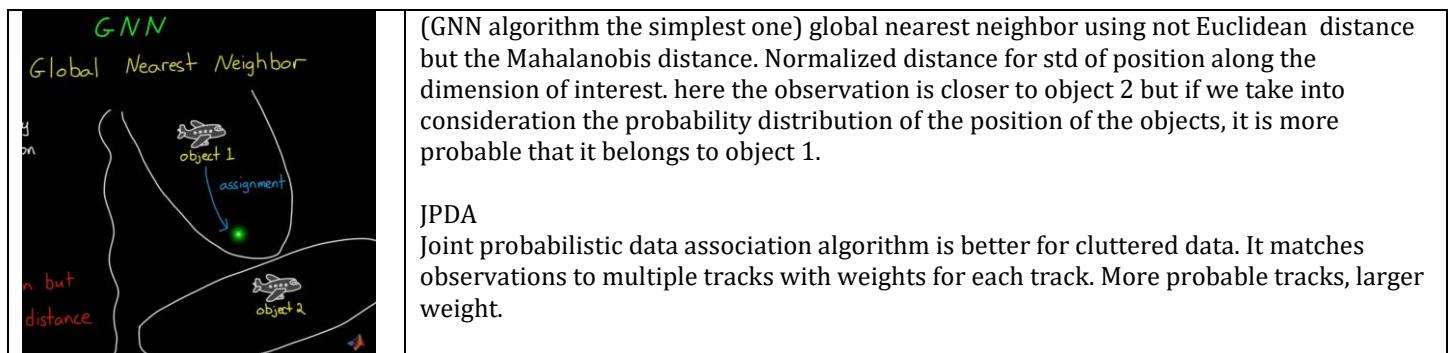
Data association is a big challenge in multitracking. We have to associate observations with tracks.

Track maintenance is another one. You have when you add a new track and when you delete one.

We can use an IMM filter to estimate the state for each tracked object.

Assignment

match an observation to a track



Track maintenance

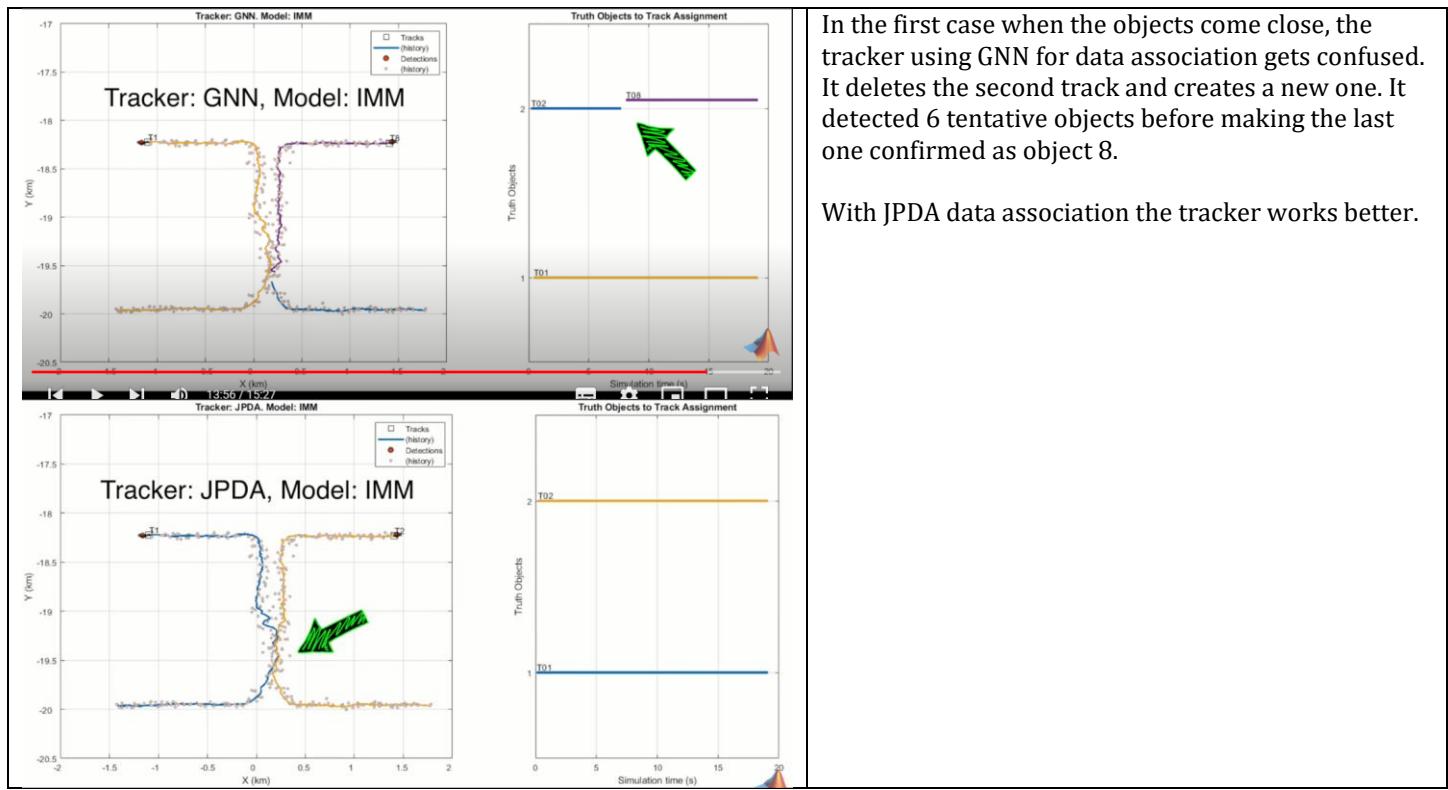
Not all observations have tracks and not all tracks have observations. How do you maintain tracks (add new tracks, delete old ones). Must be conservative.

Gating

We may choose to ignore observations outside of a defined region for each track. This is a screening mechanism that filters out observations that would not add value to our tracking. This way we make the whole tracking algorithm more efficient.

Tentative tracks

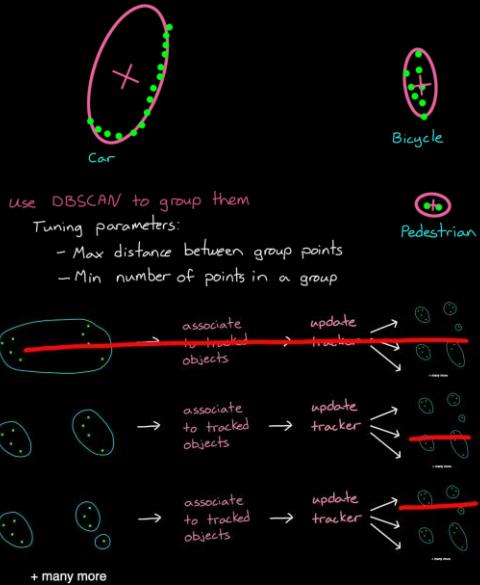
If there are new possible tracks they go to tentative state first before becoming confirmed. They need some time for additional measurements to become confirmed.



➤ Extended object tracking

This is the description of a Clustering and assignment extended object tracker. There are other methods that bypass the clustering step and directly associate each point measurement with a tracked object.

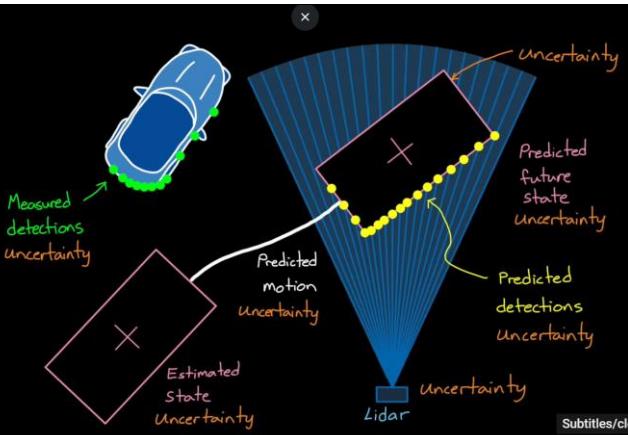
Data Association for Extended Objects



An object is large enough in relation to sensor's resolution so you have more than one data points per object. A tracker should estimate the number of objects in the scene, their position, kinematics and extend (size, shape and orientation of object). The extend can be represented with an ellipse (or a parallelogram) with 2 state variables (l_1, l_2, a). This is the partitioning problem. We have a set of point measurements, and we want to cluster them in groups of objects. But we don't know to which object a point belongs. DBScan is a clustering algorithm.

We assume every partition is possible and proceed with data association and track update for every object (group). At a later point we get new measurements. We perform new partitions and add them to the previous ones. At some point we delete the most improbable paths. At each step we must associate all tracked objects with the groups of each possible partition.

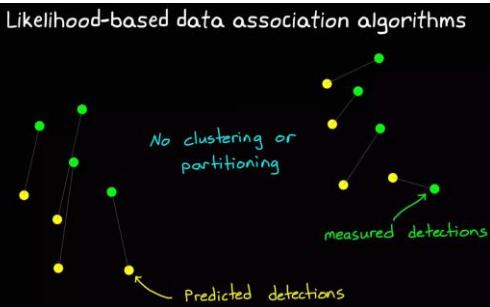
(If we have an estimated position and an estimated extend we can combine them with a Kalman filter)



Updating the tracking objects of the tracker

The tracker has an object in memory. It tracks its state (position, extend and motion model). At some point in time in the future it can make a prediction on the state of the object. It can also predict the sensor measurement at that future state based on its prediction of where the sensor will be then. The time passes, and we make a measurement of the actual object. By comparing the predicted measurement with the actual measurement, it updates this tracked object state.

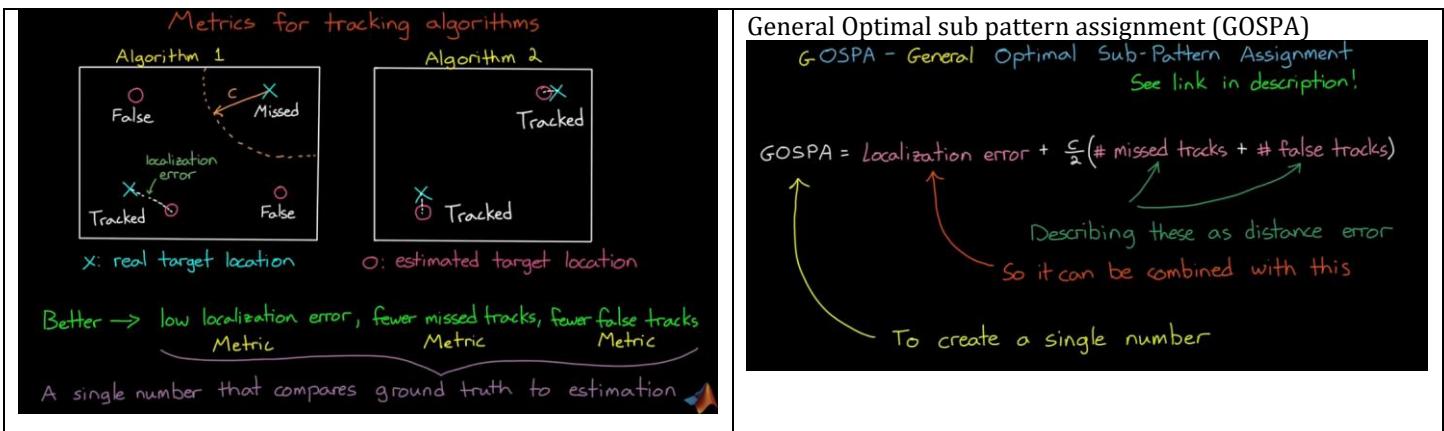
Likelihood based data association algorithms for extended object tracking



There are other methods that bypass the clustering step and directly associate each point measurement with a tracked object.

Metrics

How well your tracker does. There are more metrics like how fast it deletes or adds tracks.



Tools

- Simulation environments

mineDojo (2022) very interesting framework for autonomous agents

<https://github.com/MineDojo/MineDojo>

Autonomous agents have made great strides in specialist domains like Atari games and Go. However, they typically learn tabula rasa in isolated environments with limited and manually conceived objectives, thus failing to generalize across a wide spectrum of tasks and capabilities. Inspired by how humans continually learn and adapt in the open world, we advocate a trinity of ingredients for **building generalist agents**:

- 1) an environment that supports a multitude of tasks and goals,
- 2) a large-scale database of multimodal knowledge, and
- 3) a flexible and scalable agent architecture.

We introduce MineDojo, a new framework built on the popular Minecraft game that features a simulation suite with thousands of diverse open-ended tasks and an internet-scale knowledge base with Minecraft videos, tutorials, wiki pages, and forum discussions. Using MineDojo's data, we propose a novel **agent learning algorithm** that leverages large pre-trained video-language models as a learned reward function. Our agent is able to solve a variety of open-ended tasks specified in free-form language without any manually designed dense shaping reward. We open-source the simulation suite and knowledge bases to promote research towards the goal of generally capable embodied agents.

OpenAI gym

<https://www.gymlibrary.ml/>

Gym is a standard API for reinforcement learning, and a diverse collection of reference environments.

- Physics engines

MuJoCo

<https://mujoco.org/> (used by DeepMind control suite)

MuJoCo is a free and open source physics engine that aims to facilitate research and development in robotics, biomechanics, graphics and animation, and other areas where fast and accurate simulation is needed.

Bullet

<https://github.com/bulletphysics/bullet3>

Bullet is a physics engine which simulates collision detection as well as soft and rigid body dynamics. It has been used in video games and for visual effects in movies. This is the official C++ source code repository of the Bullet Physics SDK: real-time collision detection and multi-physics simulation for VR, games, visual effects, robotics, machine learning etc.

- Other

DeepMind Control Suite

<https://www.deepmind.com/open-source/deepmind-control-suite>

The DeepMind Control Suite is a set of continuous control tasks with a standardised structure and interpretable rewards, intended to serve as performance benchmarks for reinforcement learning agents. The tasks are written in Python and powered by the MuJoCo physics engine, making them easy to use and modify.

Misc

Robotic control

You know the mechanism of the robot so you can write down the equations of motion for it and use it as a system for which you want to find a sequence of actions for a goal. You use MPC for that. One main problem is that you train it on simulation environments. And these environments aren't perfect simulations of the world. For example, they can't simulate friction accurately. So your plan in the simulation, might not work properly in the real world. One solution would be to make some parameters of the system's motion equations being controlled by neural networks. This way, these networks could be trained in the real world and deal with any inconsistencies between real world and the simulation.

From google job post

Experience in one or more autonomous systems domains (e.g., autonomous driving, automation, digital assistants, robotic, or VR/ARs)