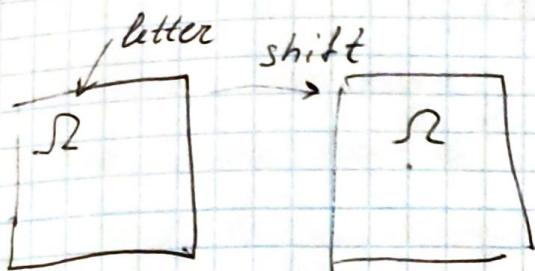
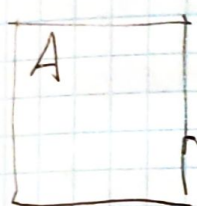


• Lemma 1 Fully-connected neural networks, automatic differentiation.



Euclidean distance is high



← Euclidean distance is lower, but letters are different.

So, raw pixels aren't a good description.

To efficiently solve ML problems, so we want to make more descriptive feature set, which we could use in linear models.

$$\{x_i, y_i\}_{i=1}^n, \quad x \in \mathbb{R}^d, \quad y_i \in \{-1, 1\}$$

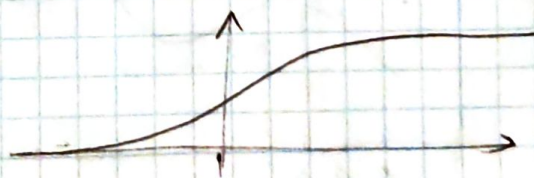
$$\text{or } y_i \in \{1, \dots, K\}$$

Simplest solution: logistic regression

$$y(x) = \text{sign}(w^T x)$$

$$\text{or } y(x) = \arg \max_{k \in \{1, \dots, K\}} w_k^T x$$

$$p(y|x, w) = \frac{1}{1 + \exp(-w^T x)}$$

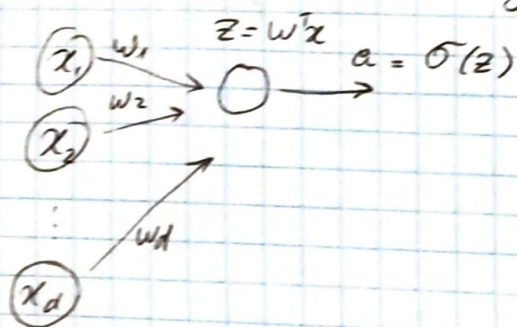


$$p(y^i|x, w) = \frac{\exp(w_i^T x)}{\sum_{j=1}^K \exp(w_j^T x)}$$

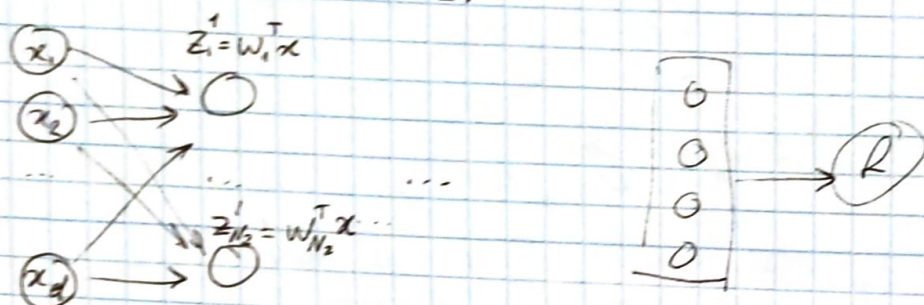
$$p(y|x, w) = \prod_{i=1}^n p(y_i|x_i, w) \rightarrow \max_w$$

$$l(w) = - \sum_{i=1}^n \log p(y_i|x_i, w) \rightarrow \min_w$$

Logistic Regression:



MLP:



$\emptyset$

$$a_i^0 = x_i, \quad i = 1, \dots, d$$

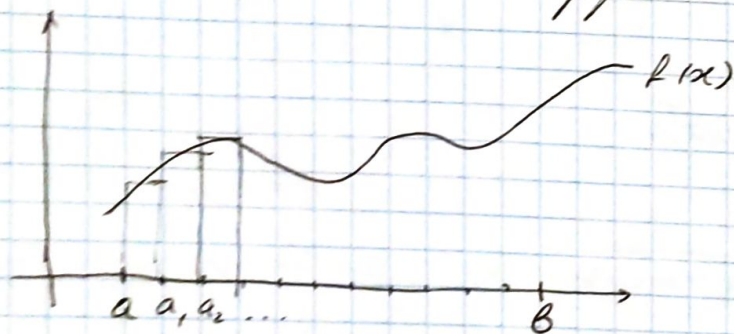
$$z^{l+1} = W^{l+1} z^l + \theta^{l+1}$$

$$a_i^{l+1} = \sigma(z_i^{l+1})$$

$$J(W) = \sum_{i=1}^n \underset{\text{loss}}{L(y_i, a^L(x_i, W, \theta))} \rightarrow \min_{W, \theta}$$

$$l \in \{1, \dots, L\}$$

MLP is a universal approximator. Consider first



$$f(x) \approx \sum_i f(a_i) [x \in [a_i, a_{i+1}]] \quad \text{it's like}$$

normal disjunctive form

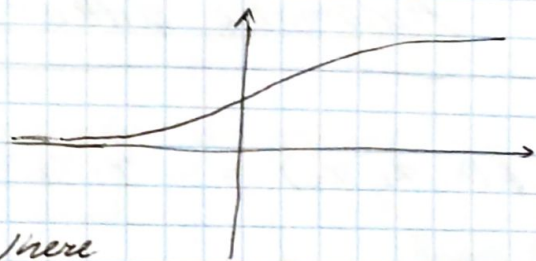
How to emulate it with sigmoids?

$$\mathbb{1}[x \geq a] \approx \sigma(k(x-a)), \quad k \gg 1$$

$$\mathbb{1}[x \in [a_i, a_{i+1}]] = [\mathbb{1}[x \geq a_i] + \mathbb{1}[x \leq a_{i+1}] > 1.5]$$

Activation functions:

① Sigmoid: $\sigma(z) = \frac{1}{1 + \exp(-z)}$



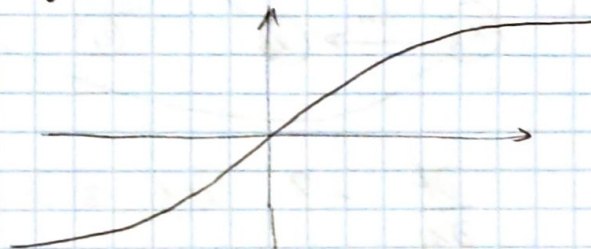
Problems: vanishing gradients where

z is far from 0. Sensitive to input only near to 0.

May be overcome by using regularization.

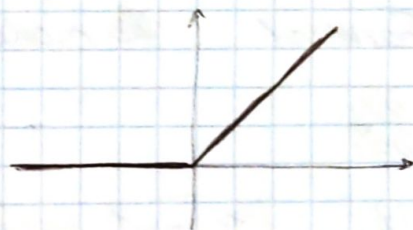
② $\tanh(z) = 2\sigma(2z) - 1$

Advantage: symmetric output region



③ $\text{ReLU}(z) = \max(z, 0)$

Advantage: cheap, sensitive to all positive z .



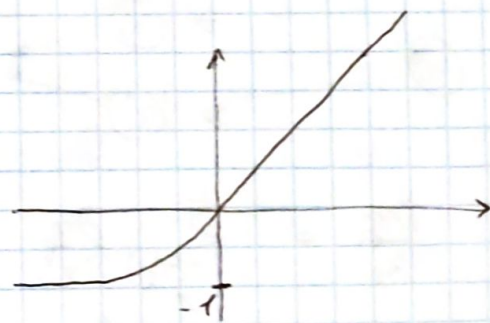
④ Leaky ReLU $(z) = \max(\alpha z, z)$

Advantage: sensitive to all z .



But it takes more iterations to optimize non-smooth functions.

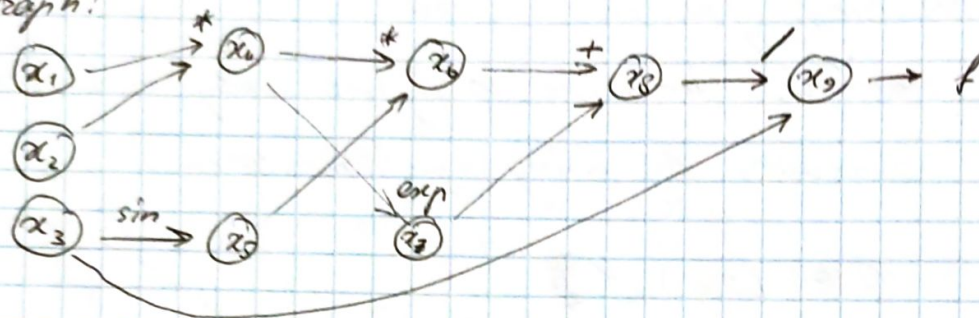
⑤ $\text{ELU}(z) = \begin{cases} z, & z \geq 0 \\ \exp(z) - 1, & z < 0 \end{cases}$



Automatic differentiation

$$f(x_1, x_2, x_3) = \frac{x_1 x_2 \sin(x_3)}{x_3} + \exp(x_1, x_2) : \mathbb{R}^3 \rightarrow \mathbb{R}$$

Comp. graph:



$$\frac{\partial f}{\partial x_i} \quad ? \quad i=1, \dots, 3$$

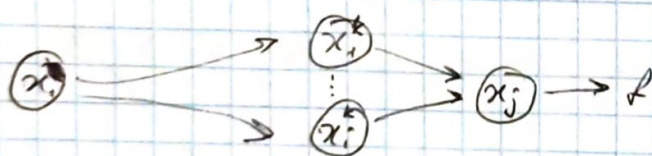
Forward mode:

$$\frac{\partial x_i}{\partial x_i}$$

$$\text{Init: } \frac{\partial x_1}{\partial x_1} = 1, \quad \frac{\partial x_2}{\partial x_1} = 0, \quad \frac{\partial x_3}{\partial x_1} = 0$$

$$x_4 = x_1 x_2, \quad \frac{\partial x_4}{\partial x_1} = \frac{\partial x_1}{\partial x_1} \cdot x_2 + x_1 \frac{\partial x_2}{\partial x_1} = x_2$$

$$x_5 = \sin(x_3), \quad \frac{\partial x_5}{\partial x_1} = \dots$$



$$\frac{\partial x_j}{\partial x_i} = \sum_{\substack{i \in \mathcal{P}(x_j) \\ \text{parent nodes}}} \frac{\partial x_j}{\partial x_i^k} \frac{\partial x_i^k}{\partial x_i}$$

This approach requires not much memory,
but need to ~~run~~ run through the graph for each var.

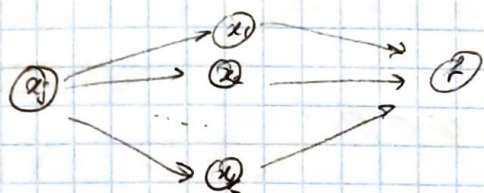
Backward mode:

On each intermediate state compute gradients of inputs with respect to the output.

$$\frac{\partial f}{\partial x_j}$$

Init: $\frac{\partial f}{\partial x_9} = 1$

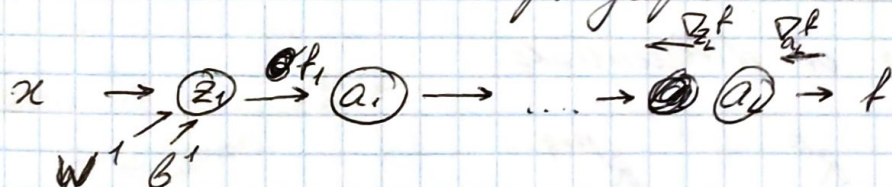
$$\frac{\partial f}{\partial x_8} = \frac{\partial f}{\partial x_9} \frac{\partial x_9}{\partial x_8} = 1 \cdot \frac{1}{x_3}$$



$$\frac{\partial f}{\partial x_j} = \sum$$

You need to store all intermediate x_i .

In case of MLP comp. graph is following:



We need to calculate $\nabla_{w_i} f, \nabla_{b_i} f$

$$f(X) = \det(A X^{-1} B) \quad \mathbb{R}^{n \times n} \rightarrow \mathbb{R}$$

$\nabla f(X)$ - want to find

$f: U \rightarrow V$, U, V - linear norm. spaces

$$\underbrace{f(x+h) - f(x)}_{\substack{\uparrow \\ \text{fixed} \\ \text{linear op. w.r.t. } h}} = d f(x)[h] + o(\|h\|)$$

$\underbrace{\quad}_{\substack{\uparrow \\ \text{fixed} \\ \text{linear op. w.r.t. } h}}$

$$d f(x)[h] : U \rightarrow V$$

List of standard differentials:

$$d(c^T X) = c^T dX$$

$$d(X^T A X) = X^T (A + A^T) dX$$

$$d(\text{tr} X) = \text{tr} dX$$

$$d \det(X) = \det(X) \text{tr}(X^{-1} dX)$$

$$dX^{-1} = -X^{-1} dX X^{-1}$$

$$d(A X B) = A dX B$$

$$d(X + Y) = dX + dY$$

$$d(XY) = dX Y + X dY$$

$$d\left(\frac{X}{\varphi}\right) = \frac{dX \cdot \varphi - X \cdot d\varphi}{\varphi^2}, \varphi \in \mathbb{R}$$

$$df(X)[dX] = d\varphi(h(X))[dh(X)[dX]]$$

$$(f = \varphi(h(X)))$$

Canonical form of differentials:

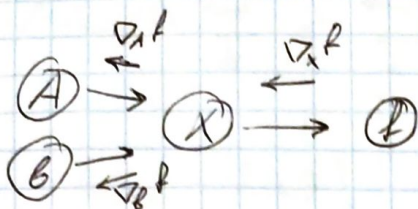
$u \backslash v$	$\mathbb{R}$	$\mathbb{R}^n$	$\mathbb{R}^{n \times q}$
$\mathbb{R}$	$f'(x)dx$	$\nabla f(x)dx$	$\nabla f(x)dx$
$\mathbb{R}^n$	$\nabla f(x)^T dx$	$\tilde{f}(x)dx$	X
$\mathbb{R}^{n \times m}$	$\text{tr}(\nabla f(x)^T dx)$	X	X

Example:

$$Ax = b$$

$$A \in \mathbb{R}^{n \times n}$$

$$b \in \mathbb{R}^{n \times 1}$$



Given: $\nabla_x f$

Find: $\nabla_A f, \nabla_b f$

$$df = \nabla_x f(x)^T dx = \text{tr} \nabla_x f^T dA + \nabla_b f^T db$$

$$Ax = b \quad | d$$

$$dAx + A dx = db$$

$$dx = A^{-1}(db - dAx)$$

~~numbers~~

$$df = \nabla_x f^T A^{-1}(db - dAx) = \nabla_x f^T A^{-1} db - \underbrace{\nabla_x f^T A^{-1} dA \cdot x}_{\text{numbers}} = \text{tr}(x \nabla_x f^T A^{-1} dA)$$

$$\nabla_b f = A^{-T} \nabla_x f$$

$$\nabla_A f = -A^{-T} \nabla_x f x^T = -\nabla_b f x^T$$

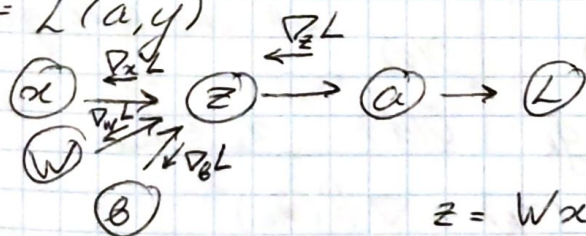
Example for MLP:

$$x \in \mathbb{R}^d$$

$$z = Wx + b, \quad W \in \mathbb{R}^{k \times d}, \quad b \in \mathbb{R}^k$$

$$a = \sigma(z)$$

$$f = L(a, y)$$



$$dL = \nabla_z L^T dz =$$

$$= \nabla_x L^T dx + \text{tr}(\nabla_W L^T dW) + \nabla_b L^T db$$

$$z = Wx + b$$

$$dz = dW \cdot x + W \cdot dx + db$$

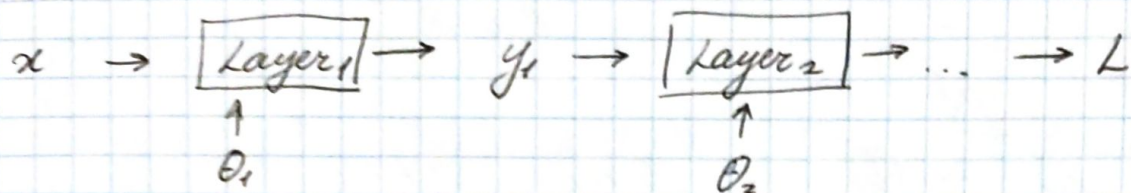
$$dL = \nabla_z L^T (dW \cdot x + W \cdot dx + db) = \text{tr}(x \nabla_z L^T dW) + \nabla_z L^T W dx + \nabla_z L^T db$$

$$\nabla_W L = \nabla_z L \cdot x^T$$

$$\nabla_b L = \nabla_z L$$

$$\nabla_x L = W^T \nabla_z L$$

In neural network



For each layer need 2 operations:

$$y_{i+1} = \text{forward}(y_i, \theta_{i+1})$$

$$\nabla_{\theta_{i+1}} L, \nabla_{y_i} L = \text{Backward}(\nabla_{y_{i+1}} L, y_{i+1}, y_i, \theta_{i+1})$$

Optimization and regularization for NN

Problem statement:

$$f(x) = \frac{1}{n} \sum_{i=1}^n f_i(x) \rightarrow \min_x, \quad n \gg 1$$

Value	Cost
$f_i(x)$	$O(q)$, $q \approx \# \text{connections in comp. graph}$
$\nabla f_i(x)$	$O(q)$
$f(x)$	$O(nq)$
$\nabla f(x)$	$O(nq)$

$$\text{SGD: } \begin{cases} i_k \sim \text{Unit}(1 \dots n) & O(1) \text{ unbiased estim.} \\ g_k = \nabla f_{i_k}(x_k) & O(q) \text{ of the full gradient} \\ x_{k+1} = x_k - \alpha_k g_k & O(d) \quad x_k \in \mathbb{R}^d \end{cases}$$

$$\mathbb{E}_{i \sim \text{Unit}(1 \dots n)} g_k = \sum_{i=1}^n \frac{1}{n} \nabla f_i(x_k) = \nabla f(x_k)$$

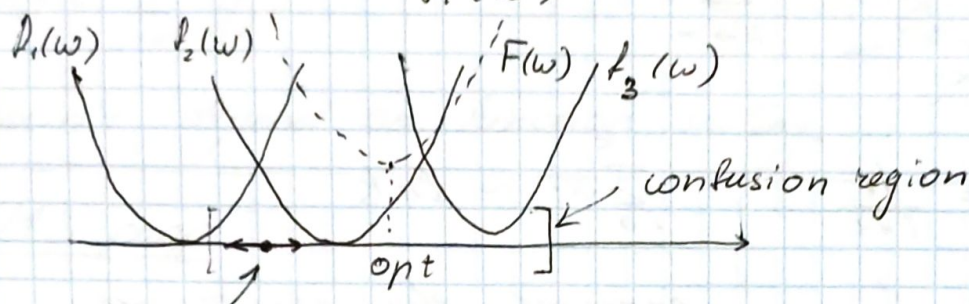
~~SGD + mini-batches~~

$$\text{SGD + mini-batches} \quad \begin{cases} J_k \sim \text{Unit}(1 \dots n) \\ g_k = \frac{1}{|J_k|} \sum_{i \in J_k} \nabla f_i(x_k) \\ x_{k+1} = x_k - \alpha_k g_k \end{cases} \quad O(|J_k|)$$

$$\text{Var } g_k \sim \frac{1}{|J_k|}$$

Example

$$F(w) = \frac{1}{n} \sum_{i=1}^n \underbrace{\frac{1}{2} (y_i - w x_i)^2}_{f_i(w)} \rightarrow \min_{w \in \mathbb{R}}$$



fluctuation in the vicinity of optimum ~~minimization~~ - no convergency to the optimal point

In classic gradient based optimization we iterate until $\|\nabla f\| < \epsilon$. But with SGD we don't have access to full gradient, so can't automatically calculate α and use gradient as stopping criteria. (In classic GD α is chosen automatically to minimize target function).

In MK we can measure quality and by that stop iter.

f - convex, $x_{k+1} = x_k - \alpha_k g_k$, $\mathbb{E} g_k = \nabla f(x_k)$

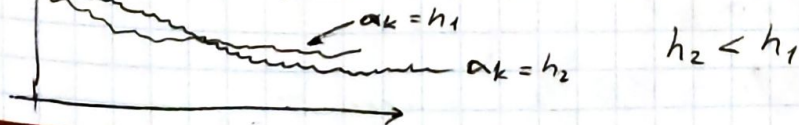
$$\boxed{\text{Th}} \quad \mathbb{E} f(\bar{x}_k) - f_{\text{opt}} \leq \frac{\|x_0 - x_{\text{opt}}\|^2 + \sum_{i=0}^k \alpha_i^2 \mathbb{E} \|g_k\|^2 \leq G^2}{2 \left(\sum_{i=0}^k \alpha_i \right)} \leq$$

$$\leq \frac{R^2 + G^2 \sum_{i=0}^k \alpha_i^2}{2 \sum_{i=0}^k \alpha_i}$$

$$\bar{x}_k = \frac{\sum_{i=0}^k \alpha_i x_i}{\sum_{i=0}^k \alpha_i} \quad \text{- mean value over all iterations with weights } \alpha_i$$

lets first consider $\alpha_i = h$

$$\mathbb{E} f(\bar{x}_k) - f_{\text{opt}} \leq \frac{R^2 + G^2 h^2 (k+1)}{2h(k+1)} = \frac{R^2}{2h(k+1)} + \frac{G^2 h}{2} \xrightarrow{k \rightarrow \infty} \frac{G^2 h}{2}$$



We want

$$\sum_i \alpha_i \rightarrow \infty, \quad \sum_i \alpha_i^2 < \infty$$

then

$$S = \frac{R^2 + G^2 \sum_{i=0}^k \alpha_i^2}{2 \sum_{i=0}^k \alpha_i} \xrightarrow{k \rightarrow \infty} 0$$

$$\alpha_i = \frac{h}{i+1}, \quad \sum_{i=0}^k \alpha_i \sim \ln k$$

$$S \sim \frac{1}{\ln k} \quad - \text{extremely slow convergence}$$

$$\text{Optimal } \alpha_i = \frac{h}{\sqrt{i+1}}, \quad S \sim \frac{1}{\sqrt{k}}$$

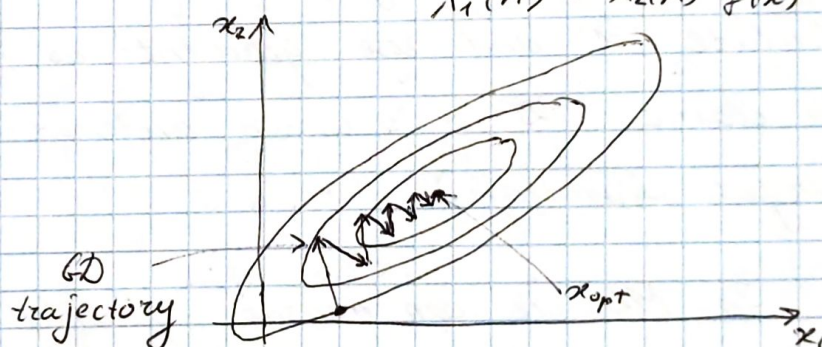
Consider $f(x) = \frac{1}{2} x^T A x - x^T b \rightarrow \min_x$

$$A = A^T > 0$$

$$\lambda_1(A) \ll \lambda_2(A) \quad f(x)$$

$$\nabla f(x) = Ax - b$$

$$\nabla^2 f(x) = A$$



GD strongly depends on scaling of optimized function.
Can't use LBFGS because it loses all of its good properties when using stochastic gradients.

SGD + momentum

$$x_{k+1} = x_k - \alpha_k g_k + \beta_k (x_k - x_{k-1})$$



$$x_1 = x_0 - \alpha g_0$$

$$x_2 = x_1 - \alpha g_1 + \beta (-\alpha g_0)$$

$$x_3 = x_2 - \alpha g_2 - \alpha \beta g_1 - \alpha \beta^2 g_0 \dots$$

Ada grad

$$x_{k+1,i} = x_{k,i} - \frac{\alpha}{\sqrt{v_{k,i} + \epsilon}} g_{k,i}$$

$$v_{k,i} = \sum_{j=0}^k g_{j,i}^2$$

Doesn't depend on function scaling.

RMS prop

$$x_{k+1,i} = x_{k,i} - \frac{\alpha}{\sqrt{v_{k,i} + \epsilon}} g_{k,i}$$

$$v_{k,i} = \beta v_{k-1,i} + (1-\beta) g_{k,i}^2$$

do exponential averaging
of the gradient

It's like Ada grad for non-convex functions

ADAM • (adaptive moments) - RMS prop + SGD momentum

$$x_{k+1,i} = x_{k,i} - \frac{\alpha}{\sqrt{v_{k,i} + \epsilon}} \mu_{k,i}$$

$$\mu_{k,i} = \beta_1 \mu_{k-1,i} + (1-\beta_1) g_{k,i}$$

$$v_{k,i} = \beta_2 v_{k-1,i} + (1-\beta_2) g_{k,i}^2$$

Usually $\alpha_{\text{start}} = 3 \cdot 10^{-4}$

Sometimes tends to overfit

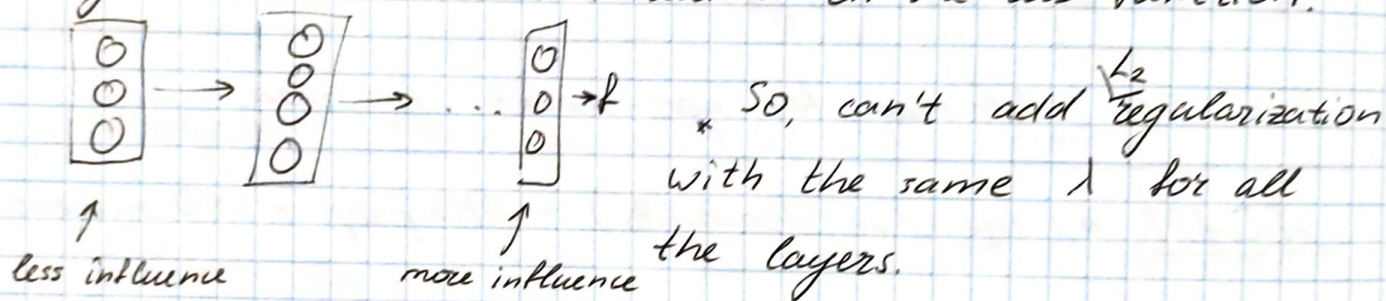
Regularization

In standard ML:

$$F(w) = \frac{1}{n} \sum_{i=1}^n L(y_i, f(w, x_i)) + \lambda \|w\|_2^2 \rightarrow \min_w$$

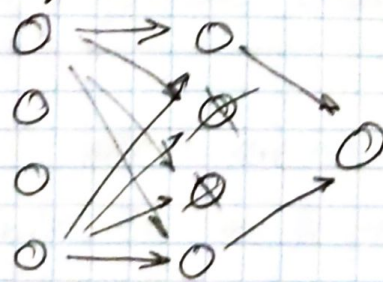
$$\begin{cases} \frac{1}{n} \sum_{i=1}^n L(y_i, f(w, x_i)) \rightarrow \min_w \\ \|w\|^2 \leq \epsilon \end{cases}$$

In case of fully-connected NN weights on different layers have different influence on the loss function.



Ideally λ should be chosen separately for each layer, but it would lead to huge amount of hyperparameters, whose tuning would require fitting the NN over and over for different sets of hyperparameters.

DropOut



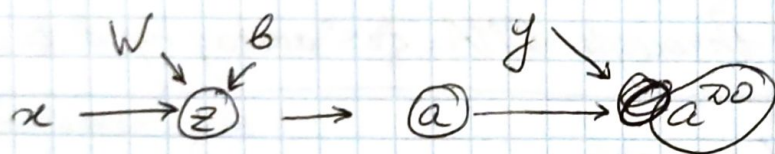
On every mini-batch on every iteration we randomly multiply outputs of neurons by 0. With probability p exclude neurons.

DropOut helps to learn all the parameters in all the layers.

$$z = Wx + b$$

$$a = g(z)$$

$$y_i \sim \text{Bern}(y_i | p): \begin{matrix} 0 & 1 \\ 1-p & p \end{matrix}$$



$$a^{\text{DO}} = y \odot a$$

$$\mathbb{E}_{y_i \sim \text{Bern}} a^{\text{DO}} = \mathbb{E} y \odot a = p a \quad \text{-- we multiply by some number, which } < 1$$

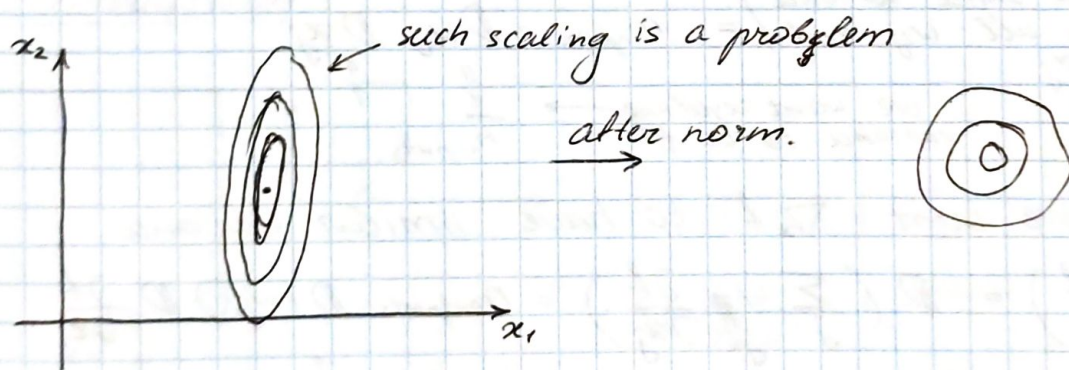
In practice

$$a^{\text{DO}} = \frac{1}{p} y \odot a \rightarrow \mathbb{E} a^{\text{DO}} = a$$

$$\text{Usually } p = \frac{1}{2}$$

On inference DropOut doesn't do anything.

Batch Normalization



On each layer we work with new feature set.

We can't compute $\mathbb{E}$ and $\mathbb{D}$ for ~~all~~ all training objects. So we compute them for batches.

$$\begin{aligned} \{x_{ij}\}_{i=1}^{N_{\text{batch}}} &\rightarrow \boxed{\text{BN}} \rightarrow \{y_{ij}\}_{i=1}^{N_{\text{batch}}} \\ \mu_j &= \frac{1}{N_{\text{batch}}} \sum_{i=1}^{N_{\text{batch}}} x_{ij} \\ \sigma_j^2 &= \frac{1}{N_{\text{batch}}} \sum_{i=1}^{N_{\text{batch}}} (x_{ij} - \mu_j)^2 \\ \hat{x}_{ij} &= \frac{x_{ij} - \mu_j}{\sigma_j + \epsilon} \\ y_{ij} &= \gamma_j \hat{x}_{ij} + \delta_j \end{aligned}$$

← optimizable params.

γ_j and δ_j are needed for compensation in case we don't need normalization.

Can't be used for small batches.

On test face μ_j and σ_j are some averaged values, obtained through EMA or averaging over one epoch.

Initialization of weights

Initialization Glorot (Xavier)

$$z = Wx$$

$$\nabla_z f \quad \nabla_x f = W^T \nabla_z f$$

$$\mathcal{D}(z_i) = \mathcal{D}\left(\sum_j w_{ij} x_j\right) = \sum_j \mathcal{D}(w_{ij}) \cdot \mathcal{D}(x_j)$$

We want zero mean and unit variance of inputs on each layer

Consider x is normalized, what's $\mathbb{E} z$ and $\mathcal{D} z$?

$$\begin{aligned} \mathcal{D} z_i &= \mathcal{D} \sum_j w_{ij} x_j = \sum_j \mathcal{D} w_{ij} \mathcal{D} x_j = \\ &\quad \text{for independent } w_{ij}, x_j \\ &= \left| \begin{array}{l} \text{the same variance} \\ \text{for all } w_{ij} \text{ and } x_j \end{array} \right| = n_{\text{inputs}} \cdot \underbrace{\mathcal{D} w_{ij}}_{\substack{\text{we want resulting} \\ \text{variance to be 1}}} \cdot \underbrace{\mathcal{D} x_j}_1 \end{aligned}$$

And we want $\nabla_x f$ to have similar variance.

$$\mathcal{D}\left(\frac{\partial f}{\partial x_i}\right) = \mathcal{D}\left(\sum_j w_{ji} \frac{\partial f}{\partial z_j}\right) = n_{\text{outputs}} \underbrace{\mathcal{D}(w_{ji})}_1 \underbrace{\mathcal{D}\left(\frac{\partial f}{\partial z_j}\right)}_1$$

To satisfy both desires, let's

$$\mathcal{D} w_{ij} = \frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}$$

Initialization strategy:

$$w_{ij}^d \sim \mathcal{N}\left(0, \frac{2}{n_{\text{in}}^d + n_{\text{out}}^d}\right)$$

d is layer index

$$\text{or } w_{ij}^d \sim \text{Unit}\left[-\frac{1}{\sqrt{m^d}}, \frac{1}{\sqrt{m^d}}\right],$$

$$m^d = \frac{n_{\text{in}}^d + n_{\text{out}}^d}{6}$$

This gives us much better

scaling for everything, compared to $\mathcal{N}(0, 1)$

Orthogonal Initialization

$W \in \mathbb{R}^{n \times p}$, $W^T W = I$ can be obtained from random matrix with Gram-Schmidt orthogonalization

In forward pass

$$z = Wx$$

$$\|z\|_2 = \|Wx\|_2 = \|x\|_2$$

Because W is orthog.

In backward pass

$$\|\nabla_x f\| = \|W^T \nabla_z f\| \neq \|\nabla_z f\|$$

unless $p = n$

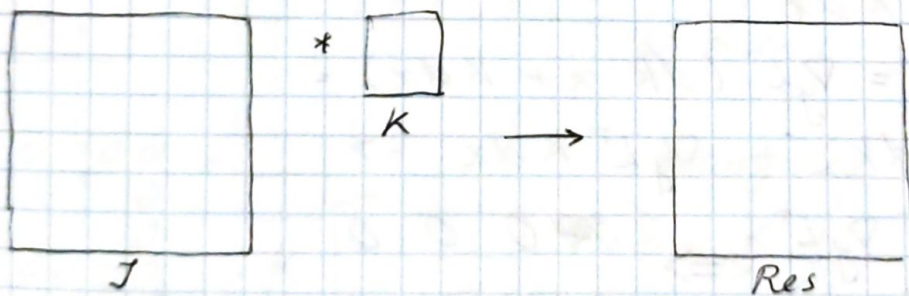
If $n \approx p$, this method is beneficial.

For RNN, for example.

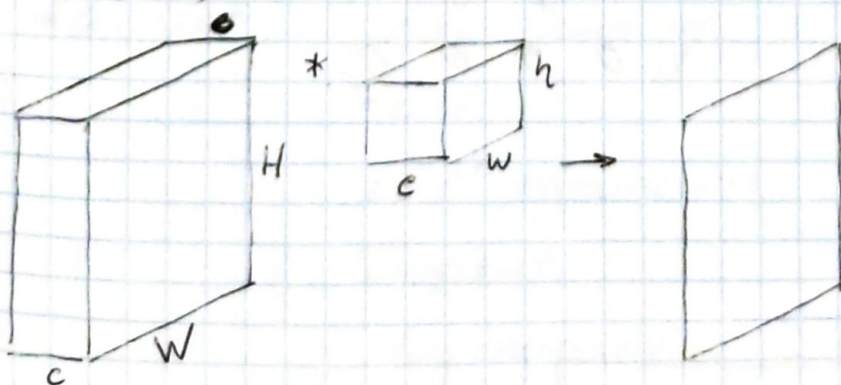
Convolutional neural networks

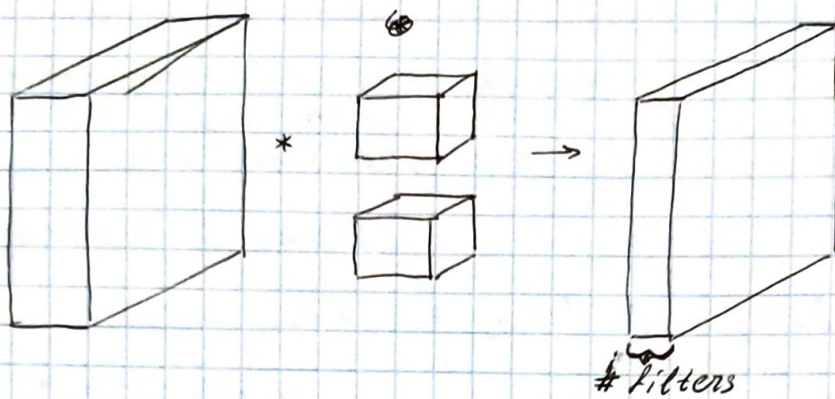
Motivation: fully conn. NN have huge number of parameters, especially in case of image processing.

Convolution



$$Res_{ij} = \sum_{u,v} K_{uv} \cdot J(i+u, j+v)$$





Backward pass

$$\begin{bmatrix} x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 \\ x_7 & x_8 & x_9 \end{bmatrix} * \begin{bmatrix} k_1 & k_2 \\ k_3 & k_4 \end{bmatrix} \rightarrow \begin{bmatrix} y_1 & y_2 \\ y_3 & y_4 \end{bmatrix}$$

$$\underbrace{\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix}}_y = \underbrace{\begin{bmatrix} \cancel{k_1} \cancel{k_2} 0 & k_3 & k_4 & 0 & 0 & 0 & 0 \\ 0 & k_1 & k_2 & 0 & k_3 & k_4 & 0 & 0 & 0 \\ 0 & 0 & 0 & k_1 & k_2 & 0 & k_3 & k_4 & 0 \\ 0 & 0 & 0 & 0 & k_1 & k_2 & 0 & k_3 & k_4 \end{bmatrix}}_K \times \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_9 \end{bmatrix}}_x$$

$$y = Kx$$

$$dy = dKx + Kdx$$

$$dL = \nabla_y L^T dy = \nabla_y L^T (dKx + Kdx) =$$

$$= \text{tr}(x \nabla_y L^T dK) + \nabla_y L^T K dx \Rightarrow$$

$$\nabla_x L = \cancel{\nabla_y L^T} K^T \nabla_y L$$

$$\nabla_k L = \nabla_y L x^T$$

$$\Leftrightarrow \begin{bmatrix} \cancel{0} & 0 & 0 & 0 \\ 0 & \nabla_{y_1} L & \nabla_{y_2} L & 0 \\ 0 & \nabla_{y_3} L & \nabla_{y_4} L & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} k_4 & k_3 \\ k_2 & k_1 \end{bmatrix} =$$

$$= \begin{bmatrix} \nabla_{x_1} L & \nabla_{x_2} L & \nabla_{x_3} L \\ \nabla_{x_4} L & \nabla_{x_5} L & \nabla_{x_6} L \\ \nabla_{x_7} L & \nabla_{x_8} L & \nabla_{x_9} L \end{bmatrix}$$

$$\boxed{\nabla_x L = \text{pad}(\nabla_y L) * \text{rev}(k)}$$

$$\nabla_K L = \nabla_y L x^T \quad \Leftrightarrow$$

$\Downarrow$

$$\nabla_K L$$

$$dK = \begin{pmatrix} dk_1 & dk_2 & 0 \\ 0 & dk_1 & \dots \\ \dots & \dots & \dots \end{pmatrix}$$

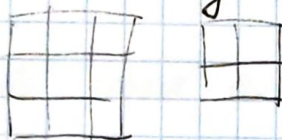
need to sum up all the same elements

$$dL = \text{tr} \nabla_K L^T dK = \nabla_K L dk$$

$$\nabla_{k_i} L = \nabla_{k_{11}} L + \nabla_{k_{22}} L + \nabla_{k_{34}} L + \nabla_{k_{45}} L$$

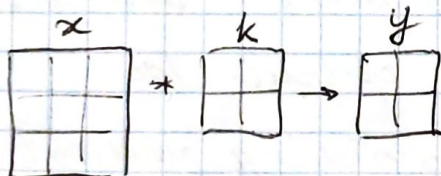
~~scribbles~~

$$\nabla_K L = x * \nabla_y L$$



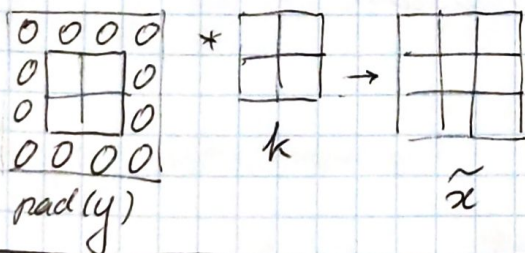
$$\nabla_K L = x * \nabla_y L$$

Convolution transposed



- forward convolution

$$y = Kx$$



- transposed convolution

$$\tilde{x} = K^T y$$

AlexNet: Max pooling

ReLU

60M parameters

Dropout

Augmentation

VGG: ~~300~~ All filters are 3×3

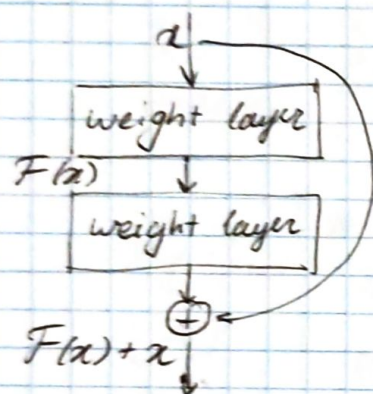
(it's cheaper to make 2 3×3 layers (18 params) than single 5×5 layer, while having the same receptive field).

140M params, 19 layers

Vanishing gradients problem

ResNet: 152 layers

Skip connections

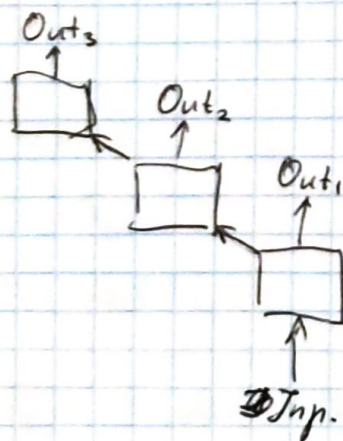
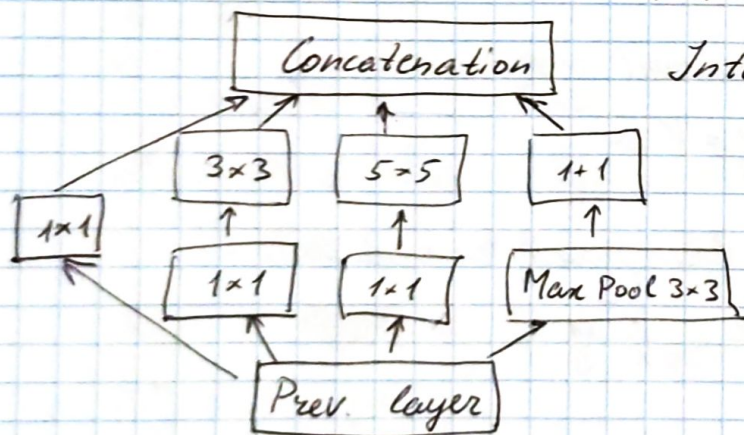


Inception:

Inception blocks

1×1 filters for dimens. reduction

Intermediate outputs for solving vanishing gradient problem



Semantic image segmentation

Task: give class label for each pixel.

~~Image classification~~

Applications:

Self-driving cars

- inference required to be fast

Medical images segmentation

Maps segmentation

etc.

May be used as first step before processing an image.

Image classification:

$$x \rightarrow NN \rightarrow \begin{matrix} y \in \{1, \dots, K\} \\ f \in \mathbb{R}^K \end{matrix}$$

$$p(y|x, \theta) = \text{softmax}(y, f(x, \theta)) = \frac{\exp(f_y)}{\sum_j \exp(f_j)}$$

$$\sum_{x, y \sim \text{Data}} -\log p(y|x, \theta) \rightarrow \min_{\theta}$$

Image segmentation:

$$x \rightarrow y \in \{1, \dots, K\}^{W \times H}, \quad |y| = K^{WH}$$

$p(y|x, \theta)$ - discrete distr. over huge domain

Energy-based models

Markov rand. field, conditional rand. field

$$p(y|x, \theta) = \frac{1}{Z(x, \theta)} \exp(-E(y, x, \theta))$$

norm. coeff. energy (should be minimized)

$$Z(x, \theta) = \sum_y \exp(-E(y, x, \theta)) - \text{can't compute directly} \\ (\text{usually use approx. algorithms})$$

$$E(y, x, \theta) = \sum_i \underbrace{\psi_i(y_i, x, \theta)}_{\substack{\text{unary potentials} \\ \text{estimation over pixels}}} + \sum_{(i,j) \in E} \underbrace{\psi_{ij}(y_i, y_j, x, \theta)}_{\substack{\text{pairwise potentials} \\ \text{estimation over neighboring pixels}}}$$

We define neighboring ~~to~~ criteria by ourselves.

$$x \in \mathbb{R}^{W \times H \times C} \rightarrow NN \rightarrow f \in \mathbb{R}^{W \times H \times K} - \text{score for each pixel} \\ \rightarrow \hat{f} \in \mathbb{R}^{W \times H \times \hat{K}} \text{ for each class}$$

$$\psi_i(y_i, x, \theta) = -\log \text{Softmax}(y_i, \underbrace{f_i(x, \theta)}_{\substack{\text{output from NN} \\ \mathbb{R}^K}})$$

Pairwise potentials ψ_{ij} introduce structural restrictions.

Classic way to calculate them:

$$\psi_{ij}(y_i, y_j, x, \theta) = \text{const} \cdot \mathbb{1}[y_i \neq y_j] \cdot (?) \\ \cdot \exp(-\frac{c}{2} \|\hat{f}_i(x, \theta) - \hat{f}_j(x, \theta)\|^2)$$

$\hat{f}_i(x, \theta)$ - additional outputs for ~~penalizing~~ structural penalizing. It may be just equal to f_i .

Training

$$\sum_{x, y \sim \text{Data}} -\log p(y|x, \theta) \rightarrow \min_{\theta}$$

Testing:

$$y = \underset{y}{\operatorname{argmax}} p(y|x, \theta) = \underset{y}{\operatorname{argmin}} E(x, y, \theta)$$

how to compute?

$$-\log p(y|x, \theta) = \log Z(x, \theta) + E(y, x, \theta)$$

$$-\nabla_{\theta} \log p(y|x, \theta) = \nabla_{\theta} \log Z(x, \theta) + \nabla_{\theta} E(y, x, \theta)$$

how to compute?

$$\nabla_{\theta} \log Z(x, \theta) = \frac{1}{Z(x, \theta)} \nabla_{\theta} Z(x, \theta) = \frac{1}{Z(x, \theta)} \nabla_{\theta} \sum_y \exp(-E(y, x, \theta)) =$$

$$= -\frac{1}{Z(x, \theta)} \sum_y \exp(-E(y, x, \theta)) \nabla_{\theta} E(y, x, \theta) =$$

$= -E_{p(y|x, \theta)} \nabla_{\theta} E(y, x, \theta)$ - can compute stochastically with some unbiased estimate of

$\hat{y} \sim \text{i.i.d. } p(y|x, \theta)$ - sample M points

$\hat{\nabla}_{\theta} \log z(x, \theta) = -\frac{1}{M} \sum_{j=1}^M \nabla_{\theta} E(\hat{y}_j, x, \theta)$ - stochastic estimate of ∇_{θ}

How to efficiently sample from this distribution?

One of the most simple:

Gibbs sampling

$y^{\text{old}} \in \mathbb{R}^D$ in our case $D = W \times H$

$y_1^{\text{new}} \sim p(y_1 | y_2^{\text{old}}, \dots, y_D^{\text{old}}, x, \theta)$ - 1-dimensional cond. distribution

$y_2^{\text{new}} \sim p(y_2 | y_1^{\text{new}}, y_3^{\text{old}}, \dots, y_D^{\text{old}}, x, \theta)$

$\vdots$

$y_D^{\text{new}} \sim p(y_D | y_1^{\text{new}}, \dots, y_{D-1}^{\text{new}}, x, \theta)$ ← here dependencies are from ψ_{ij}

y^{new}

$$p(y_i = l | y_{-i}, x, \theta) \propto p(y_i = l, y_{-i}, x, \theta) \propto$$

other elems

$$\propto \exp(-\phi_i(l, x, \theta) - \sum_{j: (i,j) \in E} \psi_{ij}(l, y_j, x, \theta))$$

Cause $\theta_{i+1} \approx \theta_i$, we can do only one Gibbs sampling per each $6D$ iteration.

How to solve problem with $\text{argmin}_y E(x, y, \theta)$ on testing face?

$$E(x, y, \theta) = \sum_i \phi_i(y_i, x_i, \theta) + \sum_{(i,j) \in E} \psi_{ij}(y_i, y_j, x, \theta) \rightarrow \min_y$$

$$y_0^1, \dots, y_0^M \sim \text{i.i.d. } p(y|x, \theta) \quad y_0^i \in \mathbb{R}^D \quad (D = W \times H)$$

$$p(y_i = l | x, \theta) = \sum_{j=1}^M \frac{\mathbb{1}[y_j^i = l]}{M}$$

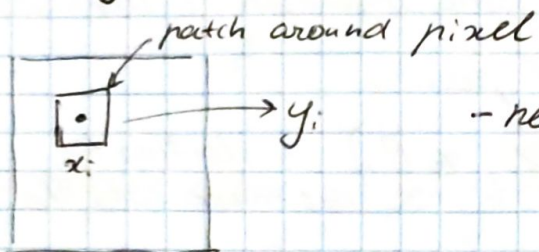
- individual marginals
Sample M segmentations,
compute pelt for each pixel,

$$y_i = \text{argmax}_{y_i} p(y_i | x, \theta) \quad \forall i \quad \text{then select class for each pixel}$$

Popular approach: $\psi_{ij} \equiv 0$

$$p(y|x, \theta) = \prod_i p(y_i | x, \theta)$$

Segmentation before NN:



- need pairwise potentials ψ_{ij}

~~It is difficult to figure out what is the correct structure~~
~~for this~~

Seg Net

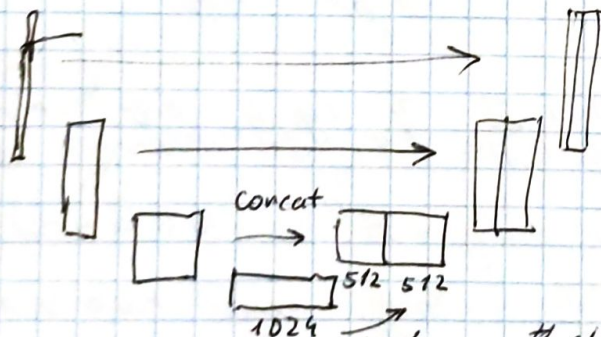
Encoder-decoder architecture

On each layer we have resolution reduction, and our task is to extract ~~initial~~ image the whole segmentation using the initial image, reduced in size. But we need to exactly find the border.

So you additionally store global information from the initial image.

We save information from max pooling operation as global data. Save positions of max pooling.

U-Net



reduce # channels with 1×1 convolution

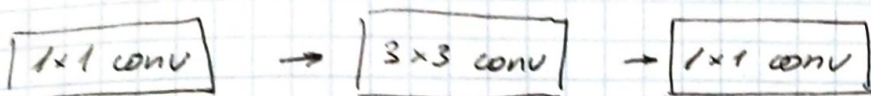
(along with LinkNet)

Is go to architecture to start from, especially for single object segmentation.

LinkNet

Instead of concatenations we do summation →
→ less parameters and faster inference.

Decoder block:



DeepLab

Uses dilated convolution in the Encoder -
- simple 3×3 conv. aren't enough to capture ~~global~~ global spatial information.

On each layer use set of differently dilated convolutions and then concatenate to the output for the next layer.

Used pairwise potentials, but it didn't give much of ~~advantage~~ advantage.

PSPNet

Variation of DeepLab.

we divide output feature map into N grids of different sizes and each cell is obtained by pooling

Uses Spatial Pyramid Pooling - ~~sum~~ on some intermediate state use poolings of different sizes, apply conv., do upsampling, concatenate and do final segmentation.

- helps to avoid large semantic errors but lacks ~~pixel~~ pixel level details.

As a Encoder we usually use some pre-trained NN (such as VGG, ResNet, etc.).

Object detection on images

We have some image and we output bounding boxes for some objects with proper class label.

Challenges:

1. Don't know a-priori how many objects are on image.

In case of single object detection we ~~make~~ can just classify the image and output additional 4 numbers : x, y, w_x, w_y .

2. There can be occlusions - objects can be only partially visible. Or there may be even several objects with the ~~same~~ same set of pixels.

(For ex., we may detect a motorbike and its wheels as separate objects)

^{so} - we can't split the image into a grid and predict only one object in each cell.

Performance / quality measure

Possible errors:

- wrong class label
- wrong b-box position
- false detection (output b-box not corresp. to any object)
- put several b-boxes on the same object
- and maybe other

MAE (mean average precision):

For position _{size}: J0.1

For classification: ~~area under~~ ^{PR} ~~ROC~~ - AUC

(~~PR~~ _{PR}: Precision = $\frac{TP}{TP+FP}$, Recall = $\frac{TP}{TP+FN}$)
y axis $\uparrow$ x axis

To make a ~~ROC~~ PR curve we take all possible thresholds. ~~PR~~

~~ROC and 1 - ROC AUC~~ and PR-AUC ~~is~~ is average precision over all thresholds for a particular class.

MAE is averaging over all possible classes.

Note, that we can't use ROC, cause there is no TN, which are background regions not detected by the model.

$$TP = \mathbb{1}[J_{OU} \geq \alpha]$$

$$FN = FP = \mathbb{1}[J_{OU} < \alpha]$$

$$AP@ \alpha = \int_0^1 \text{Precision}(\text{Recall}) d\text{Recall} = \int_0^1 P(R(p)) dR(p)$$

confidence threshold

$$mAP@ = \frac{1}{N} \sum_{i=1}^N AP_i@ \alpha$$

For each α there is a separate metric like AP50, AP75. Or we may do averaging over some set of α_i .

We calculate AP among all predictions ~~and~~ and gt-s ~~among~~ for some testing image set. If some gt box have multiple predicted boxes, intersecting with it, the one with higher J_{OU} is taken and others are ~~are~~ marked as FP.

R-CNN

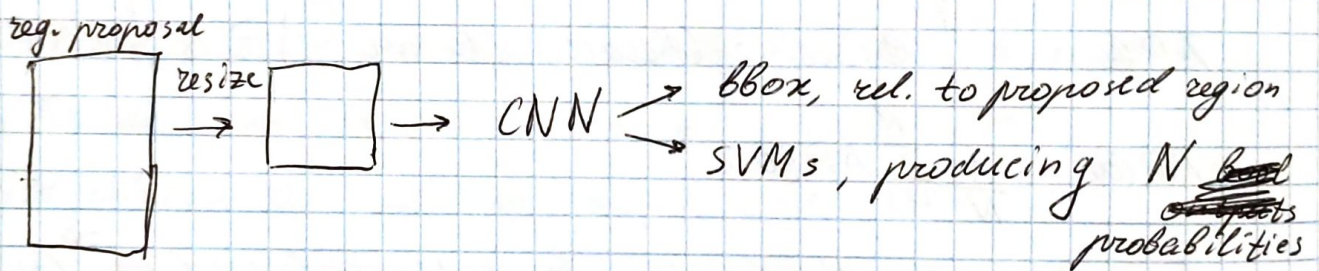
First, by some outer algorithm we extract set of bounding boxes - potential detections.

(outer algorithm in the original method is

Selective Search - some sophisticated algorithm from before NN ~~area~~ era ~~is~~ (hierarchical clustering ^{overlapping} algorithm), may output thousands of region proposals).

Then we resize each region to some standard size and feed it to classification CNN.

On the last layer we learn separate classifier for each class with answers 1 or 0.



Training:

0. Take some pre-trained CNN
1. Train it to ^{probabilities} predict $N+1$ classes ^{+1 for background} ~~via~~ through Softmax.
2. Train N SVMs on top of feature map ~~map~~ obtained at ^{some} ~~the last~~ conv. layer (we may have multiple classes for the same region).
3. Learn b-box regression coefficients on top of the same features as SVMs

Testing:

1. Predict ^(propose) k regions.
2. For each region predict b-box and class probabilities.
3. For each class make greedy non-maximum suppression
If some box has multiple classes, we ~~also~~ randomly leave only one of them.

~~greedy~~ greedy ~~non-maximum~~ non-maximum suppression:

Iterate over all class predictions and for pairs with $\text{Joll} > \text{some } \alpha$ leave bbox with higher confidence (probability). And do it in greedy way (so, depend on ordering).

Notes: b-box transformation is scale invariant and $w, h = \exp(\text{output})$ to be > 0 .

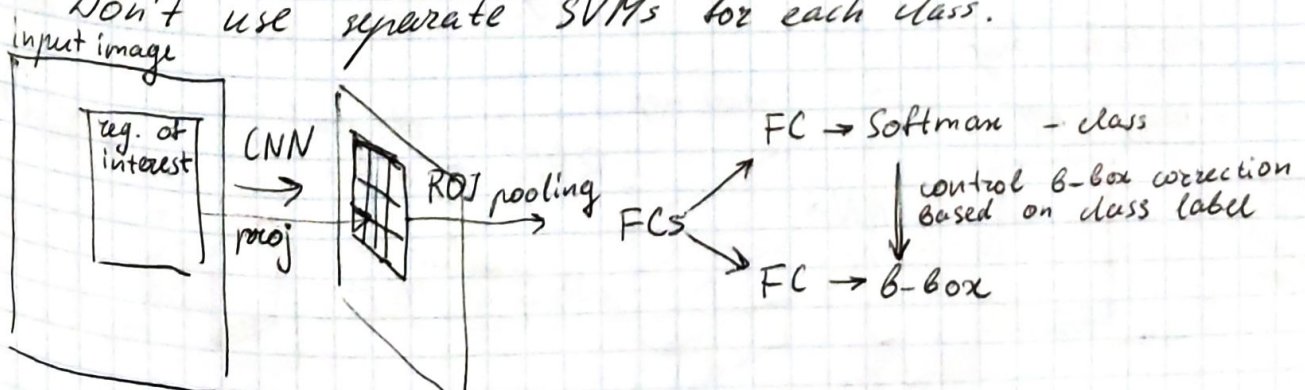
As gt b-box we probably take one with the highest Joll for our proposed region. (and get corresponding class label). If $\text{Joll} < \text{some } \alpha$, then use 'background' class label.

Fast R-CNN

We propagate our image through CNN ~~and~~ instead of it's k regions. But on ~~some level~~ some layer we project these proposed regions onto obtained feature map, then divide these regions into grids of $n \times n$ size (for some n) and do (max-) pooling in cells, projection is done in a way that each region have the same pos. ~~pos.~~ and size relative to the initial image

getting $n^2 \cdot \# \text{channels}$ features. This dividing into grid required low-level CUDA programming.

Don't use separate SVMs for each class.



Training: do in one step
Inference (testing): the same as in R-CNN

Faster R-CNN

Instead of using some external regions proposing algorithm, we learn a CNN for this task.

RPN:

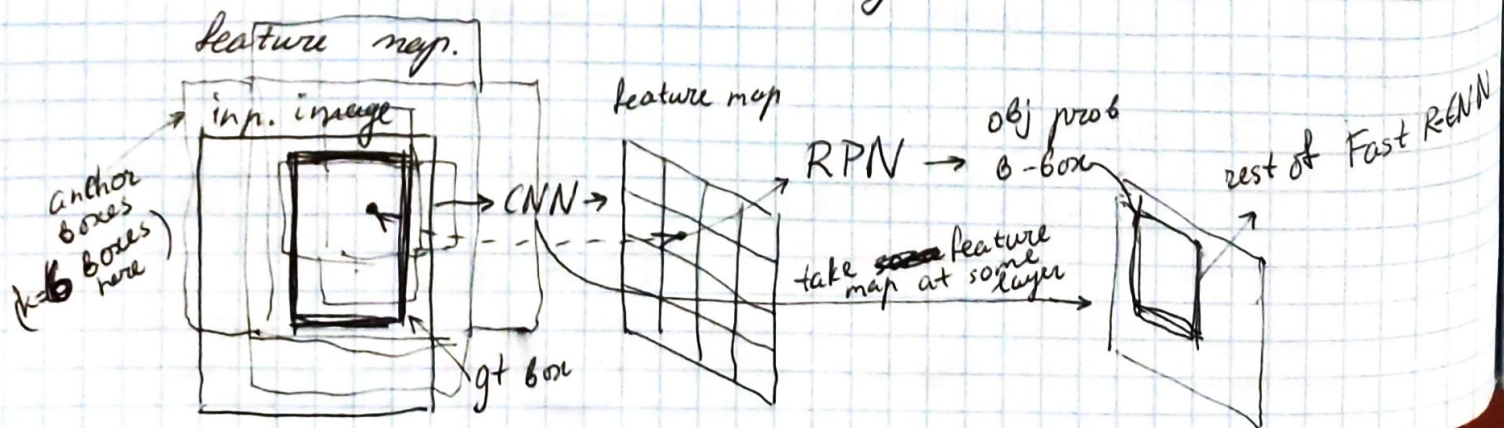
Takes feature map from last conv. layer of size $n \times n \times \text{\#ch}$
(in case of VGG-19 $n=7$, $\text{\#ch} = 512 = c$)

Then applies 3×3 conv. (or other size) to increase receptive field.

Then do 1×1 convolution with $(\text{or } 2k + 4k)$ $k \cdot b$ output channels, where k is number of regions we try to ~~prop~~ propose, and b is 1 number for background prob., 1 for prob. of some obj, 4 for b-box correction.

Each ~~proposal~~ proposal b_i among k proposal tries to fit corresponding anchor region z_i to the gt region with which it has the ~~best~~ ^{highest} IoU.

Anchor - is a box with predefined size relative to image size and with center at the same relative position into the image as it has in



For training RPN we take as positive examples ~~all~~ (for each anchor) ~~all~~ gt boxes with $\text{IoU} > 0.7$. If no anchor has such IoU , the one with highest IoU is taken. As negative we take anchors with $\text{IoU} < 0.3$. Other anchors aren't ~~updated~~ used for training (weights for their production aren't updated).

We separately train RPN and Fast R-CNN.

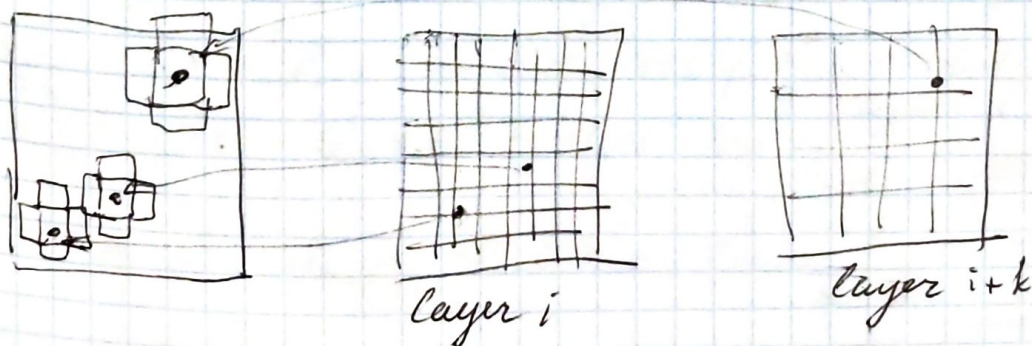
We can add ~~the~~ Feature Pyramid Network:

upscale original feature map n times (originally $n=3$),
add maps from prev layers ~~through~~ (link connections)
and instead of using $n \cdot m$ anchors for initial
feature maps, we use m anchors for each upscaled map,
(and ~~the~~ ~~we~~ apply RPN for them separately).

SSD (single shot detector)

Previous models were using 2 stage detection -
- region proposals and their classification - not fast.

Comparing to Faster R-CNN, SSD treats
anchor boxes as proposals and outputs class labels
and corrections for them as answers. We take do
this on different CNN layers, so we ~~have~~ detect
~~the~~ objects of different scales.



Retina Net

SSD + Feature Pyramid Network

For easily classified rectangles we put more weight to loss function for rare classes (and less weight for frequent) by multiplying by $(1-p_i)^{\gamma}$, where p_i is prob. of a class.

Mask R-CNN

Additionally output mask of $n \times n$ size, where $n \times n$ is resolution of feature map ~~feature~~ ^{after} RoI pooling, then upscale this mask into initial image.

Style Transfer

Input: I_{content} , I_{style}

Output: image with style applied.

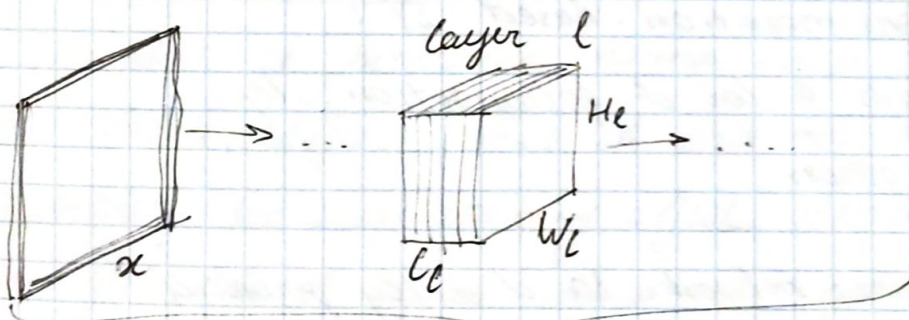
Applications: content creation, augmented reality, cartoons, games, etc.

- also data augmentation - not only change color, but also rendering method.
- also transfer learning - to adjust models to another domain (day $\rightarrow$ night, for ex.)
- also may be applied to voice, music, texts, videos

Optimization - based style transfer

First model:

VGG as a base



x_c - content image m layers

x_s - style image

x - stylized image - want to find Gatys loss

$$x = \underset{x}{\operatorname{argmin}} (L_c^l(x, x_c) + \alpha \sum_{i=1}^m w_i \cdot L_s^i(x, x_s) + \beta L_{\text{tv}}(x))$$

$L_c^l = \frac{1}{C_l} \|x^l - x_c^l\|^2$ - content loss (x^l - output tensor at layer l)

$L_s^i = \frac{1}{C_l^i \cdot W_l^i \cdot H_l^i} \|G^i(x) - G^i(x_s)\|^2$ - style loss

$G_{uv}^i(x)$ ~~and~~ $G_{uv}^i = \frac{1}{H_i \cdot W_i} \operatorname{dot}(x_{u::}^i, x_{v::}^i)$ - matrix (Gram matrix) (uncentered covariances)

Difference between content and style loss (verbally):

L_c penalizes for structural spatial differences
(tied to specific locations)

L_s is independent from particular locations

$$L_{tv}(x) = \sum_{i,j} (x_{i+1,j} - x_{i,j})^2 + (x_{i,j+1} - x_{i,j})^2 -$$

total variation loss - penalizes color differences
in neighboring pixels (to make smooth coloring).

There can be other such losses, dependent on ~~$x_{i,j}$~~

$x_{i+1,j}$, $x_{i,j+1}$, $x_{i,j}$

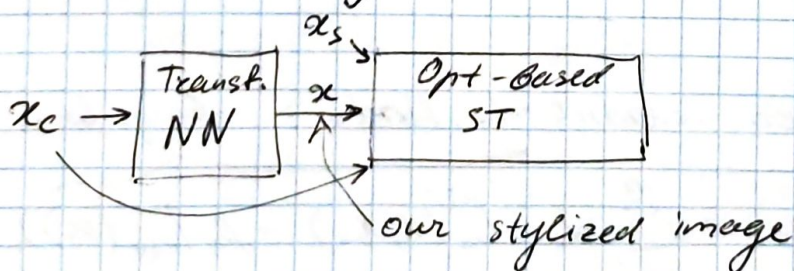
Layer l for L_c^l is chosen to preserve enough
deep features of the image, but not too abstract.

Transformation-based style transfer

Drawbacks of optimization-based ST:

- too slow (needs a lot of optimization steps)
- and maybe others

Use a transformer network to directly produce
the stylized image.



Such approach is ~~per~~ very fast, on inference,
but requires ~~separate~~ learning a separate transformer
for each style. Well-suited for video stylization.

Key properties/ ideas:

- transformer is fully-conv. NN (can be applied to image of arbitrary size)
- the transformer uses residual blocks
- (res. connections helps to propagate gradients and in initial state the network will produce ~~the~~ the same output as input)
- no pooling - stride instead (pooling removes spatial information)

With strength control:

We provided α passed into residual blocks for tuning stylization strength. On training face it's chosen randomly for each content image ~~besides~~ and used as weights for $\sum L_i$ and for res. blocks. α ~~also~~ works as regularizer.

Instance normalization

Instead of performing normalization along all batch objects, we do this normalization separately for each training object. (note, that for CNN batch norm. values are calculated among whole feature maps, and not separately for each element). In other words it's like BN with batch size = 1. And we don't need to save mean μ_i , σ_i -

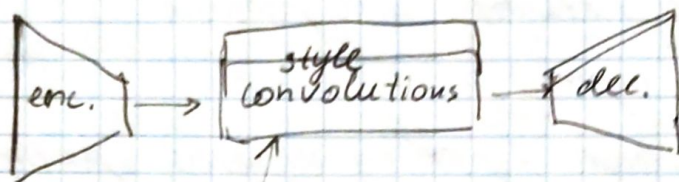
- on testing face it works the same.

Used instead BN in style transfer in more recent approaches.

Multi-style transformation-based style transfer

Style Bank

You can train k different styles



can learn this part for each needed style and then substitute for the needed one

When training we additionally train Enc. and Dec. without the style convolutions so that $D(E(x)) \approx x$ (this way we preserve spatial structure of the input image).

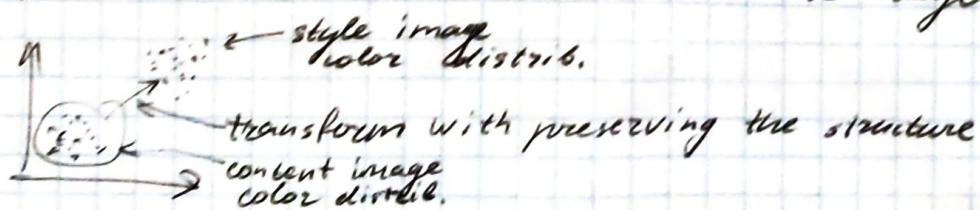
Convolution sizes may be tuned by user to produce stylization on different scales.

With this approach ~~we~~ you can mix ~~styles~~ different styles.

Also using masks you can ~~add~~ apply different styles to different parts of the image (apply them to outputs from style conv. blocks?).

Mean and variance renormalization

When transferring style we want color distribution of the input image to match content of the style image. And the same for intermediate layers.

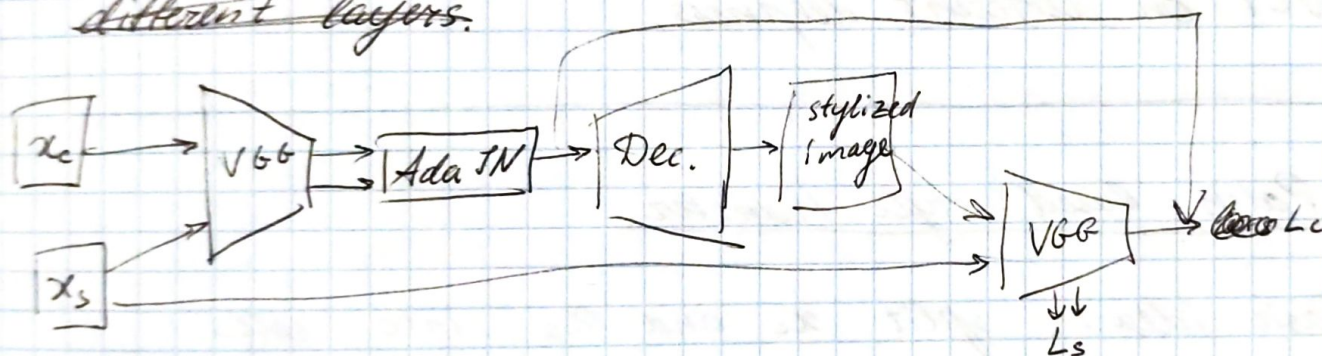


In Gatys loss it's done by matching covariance matrices $\Sigma^i(x)$. But you can do this distribution matching in your own way.

Adaptive instance normalization

Pass x_c and x_s ~~and~~ to the encoder
and on ^{output} ~~different~~ layers ^{of the encoder} do ~~style~~ transform.
to x_c^i : $x_c^i = \frac{x_c^i - \mu_c}{\sigma_c} \cdot \sigma_s + \mu_s$,
at each channel

so we match distributions of x_c and x_s on
~~different layers~~.



Can fuse styles by mixing μ_s and σ_s from
different x_s .

Doesn't reproduce low-level style details.

Extension to reproduce style details on different
levels: use AdaIN in different layers and
instead of directly taking statistics from x_s , we
tune them during learning.

For every new style you only need to fine tune
this small set of parameters $\{\mu_i^s, \sigma_i^s\}$!

You can make a network to produce these
parameters for an arbitrary style.

Mean and covariance renormalization

Whitening transformation: $\tilde{z}_c = \Sigma_c^{-\frac{1}{2}}(z_c - \mu_c) \sim F(0, I)$
it $z_c \sim F(\mu_c, \Sigma_c)$
cov matrix between channels

Coloring transformation: $\tilde{z}_c = \Sigma_s^{-\frac{1}{2}} \tilde{z}_c + \mu_s \sim F(\mu_s, \Sigma_s)$

So, we match all elements of cov. matrix, but not only diagonal.

WCT (whitening coloring transformation)

Can apply WCT style multiple times with WCT on different deepness.

Patch-based style transfer

Basic idea: split x_c and x_s into cells (patches) and for each patch in x_c replace it with most suitable patch from x_s .

With CNN we do the same but on some deep feature maps ~~after~~ ^{after} Enc. ~~feature map~~.
Then we tune x_c to fit this "stylized" ~~map~~ feature map.

$$x = \underset{x}{\operatorname{argmin}} \| \Phi(x) - \Phi(x_s) \|_2^2$$

feature map after patch matching

To preserve structure between neighboring patches of the feature map, we do patch-matching with some stride, smaller than patch size.

$$\Phi_i^{ss}(C, S) = \underset{j=1, \dots, n_s}{\operatorname{argmax}} \frac{\operatorname{dot}(\Phi_i(C), \Phi_j(S))}{\| \Phi_i(C) \| \| \Phi_j(S) \|}$$

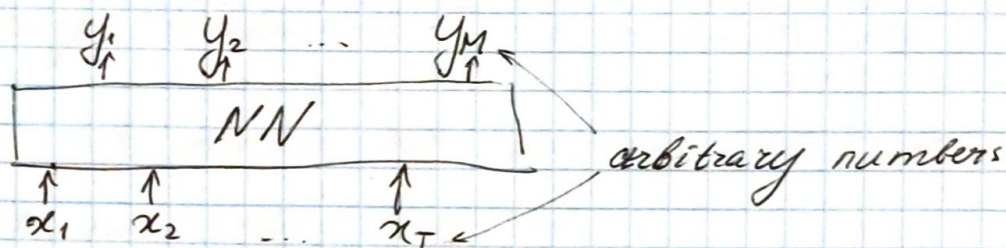
i-th patch in C
for each patch in S

- use cosine similarity

And, again, we can use ~~re~~ transformers to predict stylized image.

Recurrent NN

We deal with the following situation:



Recurrent-like NN

Loss functions:

1. $S(y) = p(y|x, \theta) = \prod_{i=1}^M p(y_i|x, \theta) \rightarrow \max_{\theta}$

$$L(y) = -\sum \log p_i$$

2. Energy-based model:

$$p(y|x, \theta) = \frac{1}{Z} \exp(-E(y|x, \theta)) = \frac{1}{Z} \exp\left(-\sum_{i=1}^M \psi_i(y_i|x, \theta) - \sum_{(i,j) \in E} \psi_{ij}(y_i, y_j|x, \theta)\right) \rightarrow \max_{\theta}$$

Usually $p(y|x, \theta) = \frac{1}{Z} \exp\left(-\sum_i \psi_i - \sum_j \psi_j(y_i, y_{j+1}|x, \theta)\right)$

For example, in NLP we would calculate ψ_i using a language model: given y_i $\psi_j = 1 - p(y_{j+1}|y_j, x, \theta)$ or just $-p(y_{j+1}|y_j, x, \theta)$ prob. is calculated according to our language model

When correspondence graph of y_j is a chain, we can efficiently compute $\arg\max_y p(y|x, \theta) = \arg\min_y E(y|x, \theta)$ using dynamic programming.

$$Z = \sum_y \exp(-E(y|x, \theta))$$

$\leftarrow$ over all possible y

When training a model:

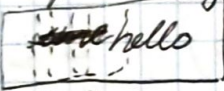
$$-\sum_{x,y \sim \text{Data}} \log p(y|x, \theta) = -\sum_{x,y \sim \text{Data}} (-\log z(x, \theta) - E(y|x, \theta)) \rightarrow \min_{\theta}$$

so we need ~~grad~~ $\nabla_{\theta} \log z(x, \theta)$ - computed using
a dynamic programming algorithm.

3. CTC loss (connectionist temporal classification)

$$\text{CTC}(y_{\text{true}}, \overset{\text{vector}}{\{p(y_j|x, \theta)\}_{j=1}^M})$$

Used, for example, in handwritten recognition:

input image
 $\rightarrow h, e, l, l, o$
 x_1, x_2, x_3

Take the image, split in possibly overlapping patches, put these patches to CNNs, get feature representations, then using RNN for ~~the~~ the obtained set of feature vectors, we produce the output text.

For the last part we should distinguish repetitive symbols (like, 'll' in 'hello') from single symbols, appearing in neighboring patches (because of patches overlapping).

So the RNN outputs something like h, e, e, ll, o and we should distinguish ee from ll .

So y_{true} is "hello" - true word - and let's consider

$$\text{y}_{\text{true}} = \text{"hello"} = D(\text{"-h-el-lo"}) \oplus$$

- so encode in a way, when adding blank "-" symbols into arbitrary positions, ^{decoding} would remain the ^{initial word} ~~word~~

$$\oplus D(\text{"hh-el-lo"}) \neq D(\text{"helo"}) = \text{"helo"}$$

- remove all ~~repetitive~~ repetitive symbols and then all "-"

Encoding: $E: D(E(y)) = y$ - so we must add "-" between duplicating symbols in y .

So, $E(y)$ is any word, ^{mandatory} where we put ~~and arbitrarily~~ in any other places ^{and} ^{and} in between ~~spaces~~ doubling symbols and then repeat each symbol 1 or more times.

We may introduce $E_M(y) = \{$

$D(y')$ is decoding of y' , which first removes all repetitions of symbols, then removes "-".
We may introduce $E_M(y) = \{y' : D(y') = y, |y'| = M\}$ ^{length of y'}

Then CTC score:

prob. distribution at j -th symbol position (prediction of our RNN)

$$CTC_s(y_{true}, \{p_j(y|x, \theta)\}_{j=1}^M) =$$

number j -th symbol of y'

$$= \sum_{y' \in E_M(y_{true})} \prod_{j=1}^M p_j(y = y'_j | x, \theta) \rightarrow \max_{\theta}$$

So, we have vector y_{true} and M discrete distributions on given set of symbols (so, it's matrix of size $M \times \#symbols_in_alphabet$).

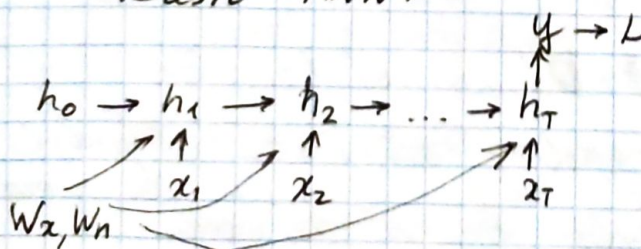
And CTC loss (logically). (seems ~~like~~ what we call CTC loss is actually CTC score, which we maximize)

$$CTC_c(y_{true}, \{p_j(y|x, \theta)\}_{j=1}^M) = -\log CTC_s \rightarrow \min_{\theta}$$

score but here sum under log

CTC ~~loss~~ is efficiently computed using dynamic programming (probably through some recurrent procedure on strings).

Basic RNN:



some activation function

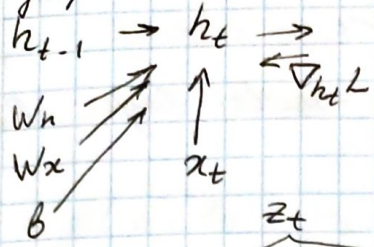
$$h_t = g(W_x x_t + W_h h_{t-1} + b)$$

W_x, W_h, b are independent from t .
 $\rightarrow$ square matrix

$$y = g(W_y \cdot h_t + b_y)$$

$$\nabla_{W_h} L = \sum_{t=1}^T (\nabla_{W_h} L)_t$$

Comp. graph at moment t :



$$h_t = g(w_x x_t + w_h h_{t-1} + b)$$

Jacobian (diag matrix in our case)

$$dL = \text{tr}(\nabla_{h_t} L^T dh_t) = \text{tr}(\nabla_{h_t} L^T \frac{\partial g}{\partial z_t} (dw_x x_t + dw_h h_{t-1} + db + w_h dh_{t-1} + db))$$

$$(\nabla_{w_h} L)_t = \frac{\partial g}{\partial z_t} \nabla_{h_t} L \cdot h_{t-1}^T$$

$$\frac{\partial L}{\partial w_h} = \sum_{t=1}^T (\nabla_{w_h} L)_t = \sum_{t=1}^T \frac{\partial g}{\partial z_t} \nabla_{h_t} L \cdot h_{t-1}^T$$

$$\boxed{\nabla_{w_h} L} = \sum_{t=1}^T (\nabla_{w_h} L)_t = \sum_{t=1}^T \frac{\partial g}{\partial z_t} \nabla_{h_t} L \cdot h_{t-1}^T =$$

$$= \sum_{t=1}^T \frac{\partial g}{\partial z_t} w_h^T \frac{\partial g}{\partial z_{t+1}} \nabla_{h_{t+1}} L h_{t-1}^T =$$

$$= \sum_{t=1}^T \frac{\partial g}{\partial z_t} \left(\prod_{j=t+1}^T w_h^T \frac{\partial g}{\partial z_j} \right) \nabla_{h_t} L \cdot h_{t-1}^T$$

grad. is proportional to some prod. over same matrix

$$\|\nabla_{w_h} L\| \leq \sum_{t=1}^T \left\| \frac{\partial g}{\partial z_t} \right\| \left(\prod_{j=t+1}^T \|w_h^T \frac{\partial g}{\partial z_j}\| \right) \cdot \|\nabla_{h_t} L \cdot h_{t-1}^T\|$$

If $\|w_h^T \frac{\partial g}{\partial z_j}\| < 1 \quad \forall j$, $\Rightarrow$ gradient vanishing problem towards first iterations (can't learn long dependencies)

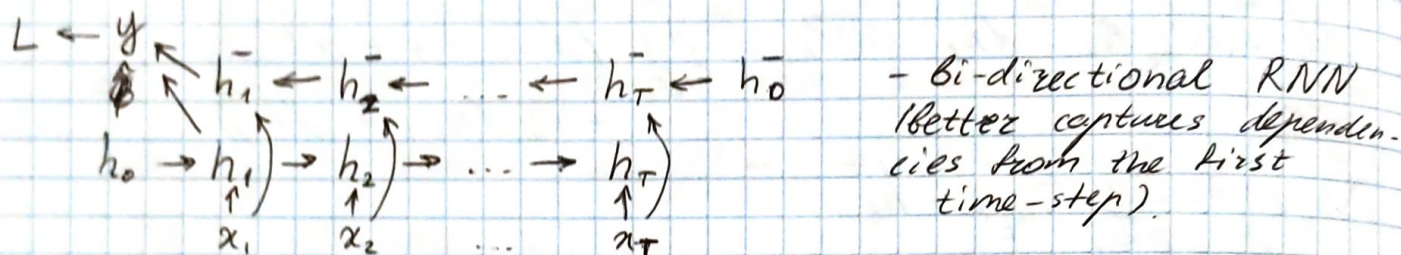
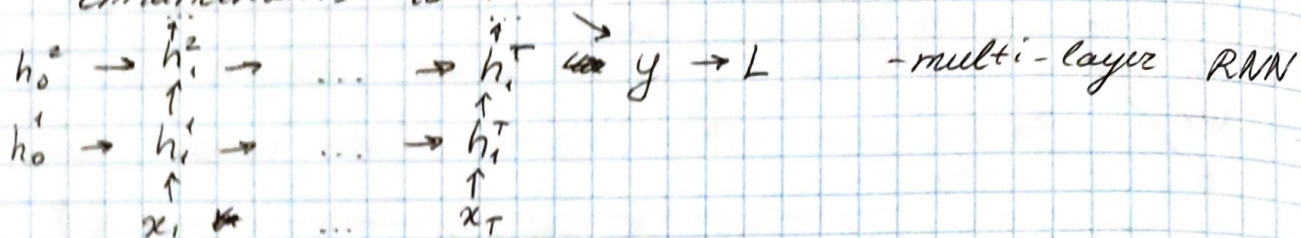
If $\|w_h^T \frac{\partial g}{\partial z_j}\| > 1 \quad \forall j \Rightarrow$ gradient exploding problem

FC / CNN

RNN

- | | |
|--------------------------------|-------------------------------------|
| 1. Residual (skip) connections | $\rightarrow$ LSTM / GRU |
| 2. Batch normalization | $\rightarrow$ Layer normalization |
| 3. Drop Out | $\rightarrow$ DropOut (special one) |
| 4. Weights initialization | $\rightarrow$ Weight norm. |

Enhancements to the basic structure:



Weights initialization:

For $W_h \in \mathbb{R}^{m \times m}$ want to be orthogonal: $W_h^T W_h = W_h W_h^T = I$,

because $\|W_h\|_2 = \|I\|_2$

And we want W_h to be orthogonal on each training iteration (and on inference). ~~the~~

We may preserve this property by computing W_h from another matrix:

$$\text{If } X = -X^T, \quad W = \expm(X) \Rightarrow W^T W = W W^T = I$$

$\uparrow$
matrix

$= L + -L^T$ for arbitrary ~~matrix~~ L

$\det(W) = 1$ for such construction

$W_h = \expm(\tilde{W}_h - \tilde{W}_h^T)$ - and we need to compute it only once for all time moments

LSTM (long-short term memory network)

Consider 15. * abc daed # $\rightarrow$ aa...a

$\uparrow$ some number $\nwarrow$ any sequence $\nwarrow$ would like to learn

$\underbrace{\hspace{1.5cm}}_{15 \text{ times}}$

Idea: make an RNN, which can at each time moment store some data to memory and load some data from memory and forget non-needed data.

LSTM cell:

h_t - "short-term memory"

c_t - "long-term memory"

$$g_t = g(W_x x_t + W_h h_{t-1} + b) \quad \text{- as in basic RNN}$$

$$i_t = \sigma(W_x^i x_t + W_h^i h_{t-1} + b^i) \in (0, 1) \quad \text{- "input gate"}$$

$$o_t = \sigma(W_x^o x_t + W_h^o h_{t-1} + b^o) \in (0, 1) \quad \text{- "output gate"}$$

$$f_t = \sigma(W_x^f x_t + W_h^f h_{t-1} + b^f) \in (0, 1) \quad \text{- "forget gate"}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t \quad \leftarrow \text{kind of residual connection}$$

$$h_t = o_t \odot \tanh(c_t)$$

$\leftarrow$ ~~we~~ tanh needed because c_t can accumulate large numbers over iterations

f_t for reading from memory and for forgetting (erasing)

i_t for writing new data into memory

o_t for outputting needed data from memory

GRU (gated recurrent unit network)

An attempt to reduce the number of parameters in LSTM.

Have only 2 gates:

$$r_t = \sigma(W_x^r x_t + W_h^r h_{t-1} + b^r) \quad \text{- "reset gate"}$$

$$u_t = \sigma(W_x^u x_t + W_h^u h_{t-1} + b^u) \quad \text{- "update gate"}$$

$$h_t = \cancel{K} (1 - u_t) \odot h_{t-1} + u_t \odot g_t \quad \text{- current information}$$

$$g_t = g(W_x x_t + W_h (h_{t-1} \odot r_t) + b)$$

g_t overwrites prev. inform.

Have one less gate and one less state.

Initialization: $b^f \gg 0$ in order to f_t not erase information on the first steps

and $b^i \gg 0$ to use g_t

For GRU: $b^u = 0$? (in the lecture $b^u \gg 0$)
 $b^r \gg 0$ $\leftarrow$ to be able to read and write to h_t

Layer normalization

Batch norm.: $\mu_j = \frac{1}{|B|} \sum_{i \in B} h_{tj}(i)$

$$h'_{tj} = h_{tj}(i) - \mu_j$$

But $|B|$ is too small in ~~one~~ case of RNN.

Layer norm.: ~~BN~~, ~~where~~ but we treat ~~the~~ recurrent layer inputs over time moment as objects of a batch.

$$\mu_j = \frac{1}{T} \sum_{k=1}^T h_{kj}, \quad \sigma_j = \dots$$

$$h'_{kj} = \frac{h_{kj} - \mu_j}{\sigma_j}$$

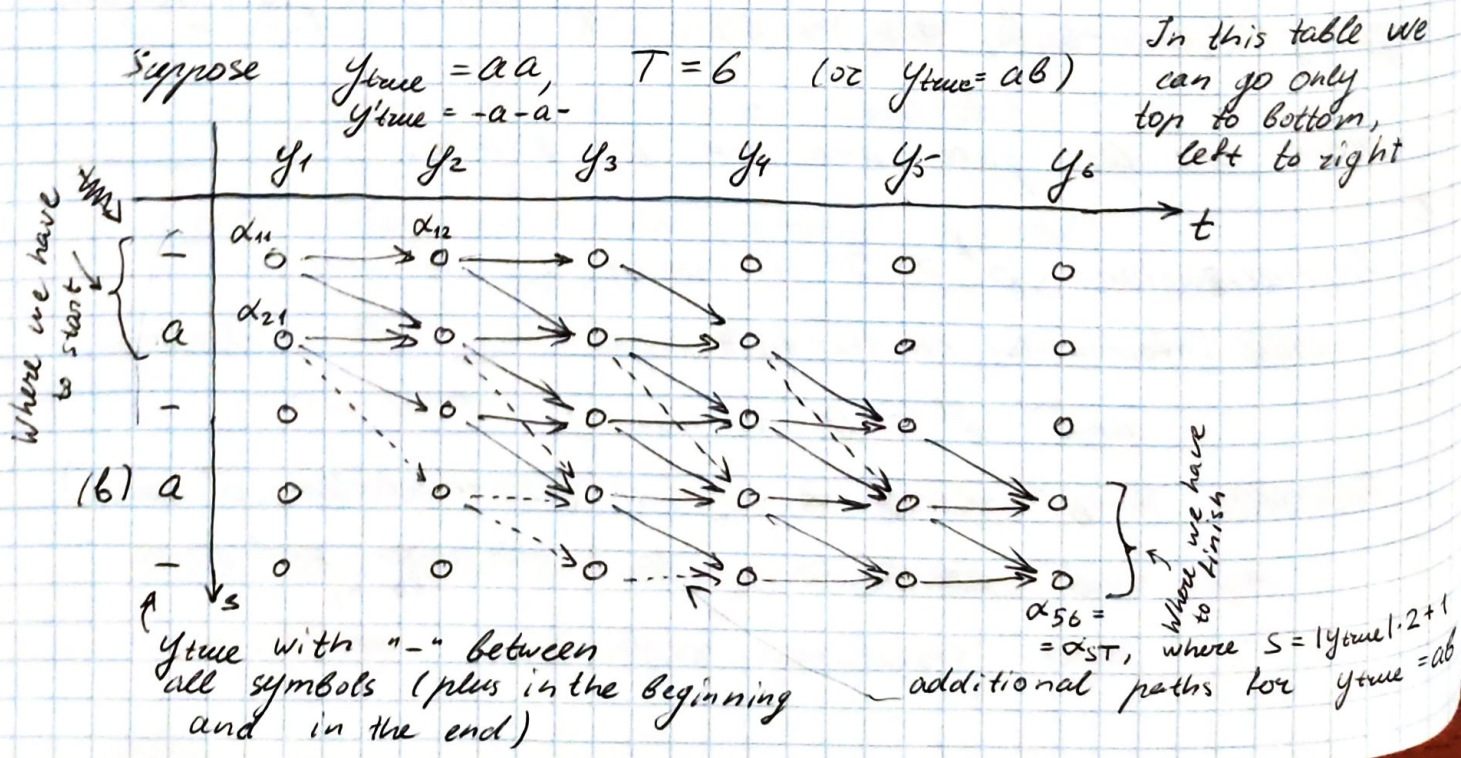
Drop Out

Naive implementation produces zero gradients ~~at~~ towards beginning of a sequence (if we put ~~the~~ dropout on the ~~recurrent~~ recurrent connections).

For LSTM: $c_t = f_t \cdot c_{t-1} + i_t \cdot d(q_t)$ ← Dropout

For GRU: $h_t = (1 - u_t) \cdot d(q_t) + u_t h_{t-1}$

CTC loss computation (which is CTC score)



~~How to compute?~~

How to compute?

$$CTC_s(y_{true}, x, \theta) = \sum_{y: CTC(y) = y_{true}} \prod_{t=1}^T p_t(y = y_t | x, \theta)$$

Let α_{st} be accumulated prob. at time moment t at symbol with index s of our extended y_{true} .

(see the table). And let $y_{true}^{[s]}$ be symbol of y_{true} at s position

$$\alpha_{st} = (\alpha_{s-1, t-1} + \alpha_{st-1} + \mathbb{1}(y_{true}^{[s]} \neq y_{true}^{[s-2]}) \cdot \alpha_{s-2, t-1}) \cdot$$

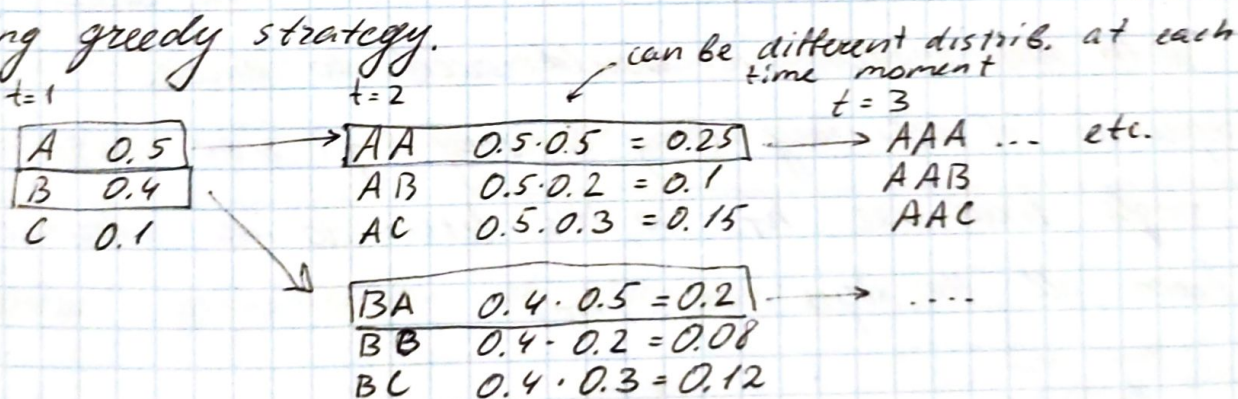
$$p(y_{true}^{[s]} = y_t | x, \theta)$$

$$CTC_s = \sum_{i=0}^{T-S} \alpha_{s-i, T}, \quad S = 2|y_{true}| + 1 = |y_{true}'|$$

Beam search

~~Generalization~~ Generalization of greedy search algorithm, & when we search N best paths (instead of 1)

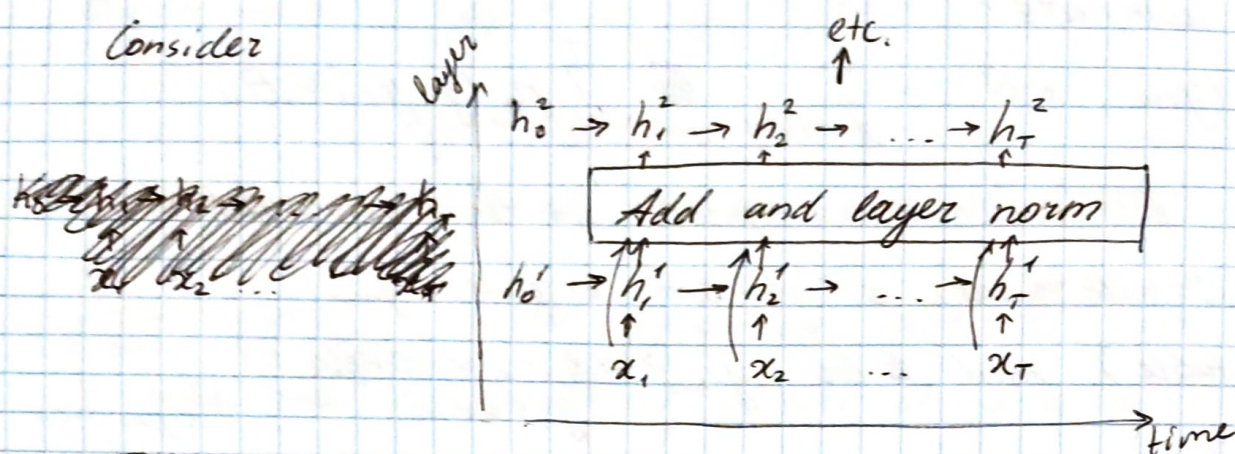
using greedy strategy.



Beam search ~~search~~ of 2 best paths in alphabet $\{A, B, C\}$.

Attention mechanism

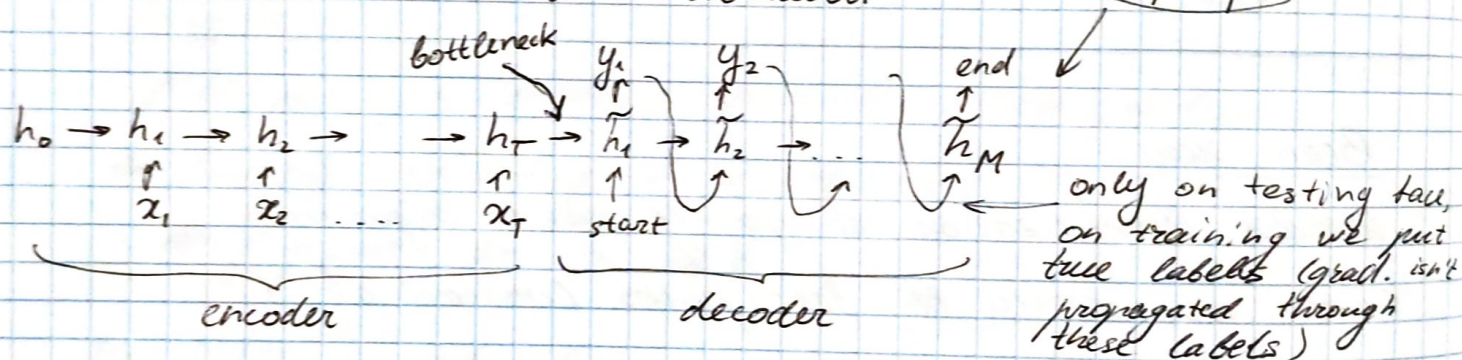
Consider



These models aren't good at transforming some sequence into another sequence.

For example, copy task

aebaabd $\rightarrow$ aebaabd aebaabd



With such model we would need to encode sequences of arbitrary (may be very big) size into single fixed-size h_T . So, it's better to use information from all the steps $h_1, \dots, h_T$.

Attention mechanism:

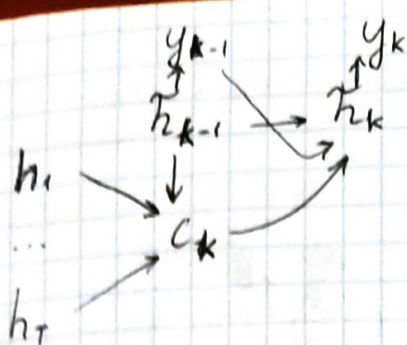
Consider $\tilde{h}_{t-1}$ and we need to gen $\tilde{h}_t$ and y_t .

Similarity measure: $\text{sim}(\tilde{h}_{t-1}^k, h_t) \in \mathbb{R}$

$$\alpha_t^k = \frac{\exp(\text{sim}(\tilde{h}_{t-1}^k, h_t))}{\sum_{j=1}^T \exp(\text{sim}(\tilde{h}_{t-1}^k, h_j))}, \quad t \in \{1, \dots, T\}$$

$$\alpha^k = \text{Softmax}(\text{sim}(\tilde{h}_{t-1}^k, h))$$

$$c_k = \sum_{j=1}^T \alpha_j^k h_j = \sum_{t=1}^T \alpha_t^k h_t$$



- computation graph at time moment k during decoding pass

Seq2seq with attention

Ideas for computing $\text{sim}(\tilde{h}_k, h_t)$:

1. $\text{sim}(a, b) = a^T b$, if $\dim(a) = \dim(b)$
2. $\text{sim}(a, b) = \frac{a^T b}{\sqrt{\dim(a)}}$, if $\dim(a) = \dim(b)$

if $a_i \sim N(0, 1)$ and $b_i \sim N(0, 1)$, then

$$E\left(\frac{a^T b}{\sqrt{\dim(a)}}\right) = E\left(\frac{\sum a_i b_i}{\sqrt{\dim(a)}}\right) = 0$$

$$D\left(\frac{a^T b}{\sqrt{\dim(a)}}\right) = \frac{1}{\dim(a)} \sum_i D a_i^2 D b_i^2 = 1$$

So, such approach gives good scaling.

3. $\text{sim}(a, b) = W_3^T \tanh(W_1 a + W_2 b)$ for arbitrary $\dim(a)$, $\dim(b)$
 additional params to train

Suitable for cases, when input and output sequence are in different domains.

General representation of attention:

$$c = \text{Att}(q, K, V)$$

$q \in \mathbb{R}^{\dim(q)}$ - query

$K = \{k_i\}_{i=1}^T$, $k_i \in \mathbb{R}^{\dim(q)}$ - keys

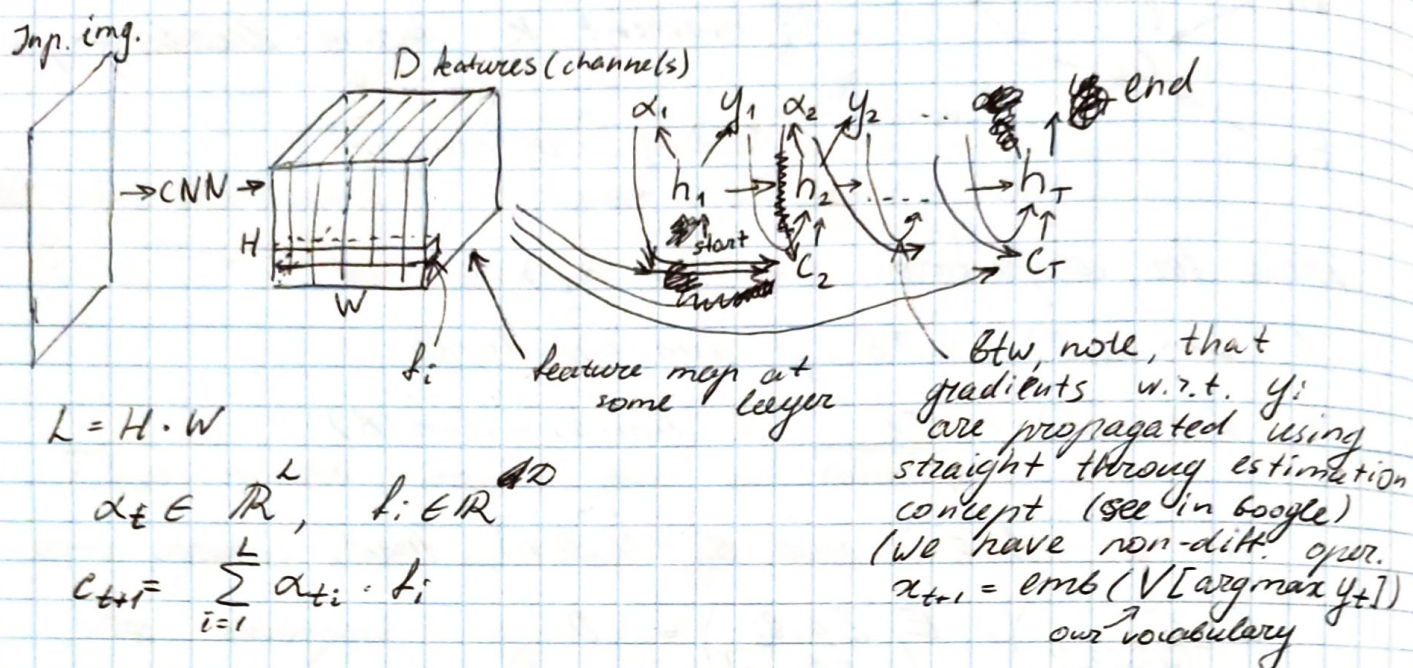
$V = \{v_i\}_{i=1}^T$, $v_i \in \mathbb{R}^{\dim(v)}$ - values

note, that $\dim(k_i) = \dim(q)$

$$\alpha_t = \frac{\exp(\text{sim}(q, k_t))}{\sum_i \exp(\text{sim}(q, k_i))}, \quad \alpha = \text{Softmax}(\text{sim}(q, K))$$

$$c = \sum_{t=1}^T \alpha_t v_t$$

Image captioning



Teacher forcing + exposition bias

Teacher forcing: on training face to compute h_t we use $y_{\text{true}}[t-1]$ instead of y_{t-1} (our prev. prediction). But it introduces exposition bias (model doesn't learn to use previous predictions). So, with some probability we put y_{t-1} after all.

Beam search

(see on prev pages)

Use it on inference.

Byte-Pair Encoding

Using words as tokens would make them too big (Softmax would have comp. problems) because vocabulary is too big. And we couldn't learn statistics for rare words.

On the contrary, using only symbols would make it hard to learn words and some correct syntax.

So, it's better to use BPE tokenization:

aa bd aa bc a → Aa bd Aa bc a → BBd BBca →

aa - 4 → A
ab - 2
bd - 1
....

Aa - 2 → B
ab - 2 ~~4~~
...

BB - 2 → C
bd - 1
....

→ C d l c a - continue, until we get N tokens (N is set by user).
or until we don't have repeating pairs.
These tokens are then used instead of words.

To construct correct words on inference, we need to use some language model in addition to beam search (LM would output prob. for the next token, given current one, and we add log of this prob.)

There can be different implementations.

(we also add special symbol at the ends of words)

BLEU - CTDER optimization

BLEU and CTDER are some metrics to evaluate the score for translated / generated text given set of gt texts.

Our goal is to minimize: $F(\theta) := \mathbb{E}_{p(y|x, \theta)} f(y, x)$ $\rightarrow \min_{\theta}$ non-differentiable

$$\nabla_{\theta} F(\theta) = \nabla_{\theta} \sum_y \overset{\text{big sum}}{p(y|x, \theta)} f(y, x) = \sum_y f(y, x) \nabla_{\theta} p(y|x, \theta) =$$
$$= \left| \nabla_{\theta} \log p(y|x, \theta) = \frac{\nabla_{\theta} p(y|x, \theta)}{p(y|x, \theta)} \right| =$$

$$= \sum_y p(y|x, \theta) (\nabla_{\theta} \log p(y|x, \theta)) \cdot f(y, x) =$$

$$= \mathbb{E}_{p(y|x, \theta)} f(y, x) \nabla_{\theta} \log p(y|x, \theta) \quad \text{can do Monte Carlo estimation with Gibbs sampling}$$
$$\approx \frac{1}{M} \sum_{j=1}^M f(y_j^i, x) \nabla_{\theta} \log p(y_j^i|x, \theta), \quad y_j^i \sim p(y|x, \theta)$$

But we have here a large variance.

So, instead we calculate

$$\ominus \mathbb{E}_{p(y|x, \theta)} \nabla_{\theta} \log p(y|x, \theta) (f(y, x) - b(x))$$

(why we can add $b(x)$):

$$\ominus \mathbb{E}_{p(y|x, \theta)} \nabla_{\theta} \log p(y|x, \theta) f(y, x) - \underbrace{\mathbb{E}_{p(y|x, \theta)} \nabla_{\theta} \dots}_{(*)} \cdot b(x)$$

$$(*) = \sum_y \nabla_{\theta} p(y|x, \theta) \cdot b(x) = b(x) \cdot \nabla_{\theta} \sum_y p(y|x, \theta) = b(x) \cdot \nabla_{\theta} 1 = 0$$

Possible choice for $b(x)$:

$$b(x) = \mathbb{E}_{p(y|x, \theta)} f(y, x)$$

- but needs a lot of samples on each iteration for each object in a batch

Alternative way:

$$b(x) = f(y_{\text{greedy}}, x)$$

y_{greedy} - one sample obtained from output distribution by greedy strategy

Such optimization is used for fine-tuning on the last stage of training

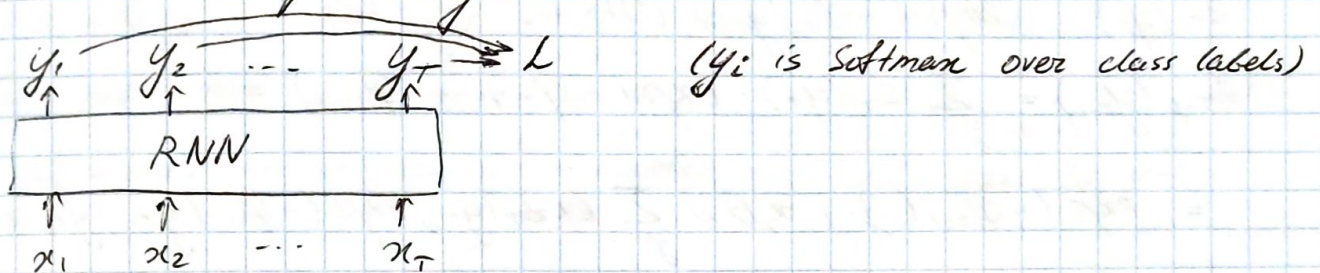
Natural language processing

Text segmentation problem

Examples: words tagging ("verb", "noun", etc.),
entity recognition (addresses, names, etc.).

Task: for each input token output its class label

The most simple way to solve:



$$p(y|x, \theta) = \frac{1}{Z} \exp(-E(y, x, \theta))$$

$$E(y, x, \theta) = \sum_{i=1}^T \varphi_i(y_i, x, \theta) + \sum_{i=1}^{T-1} \psi_i(y_i, y_{i+1}, x, \theta)$$

$$Z = \sum_y \exp(-E(y, x, \theta)) \text{ - huge sum}$$

$$\text{Optimization task: } L = - \sum_{x, y \sim \text{Data}} \log p(y|x, \theta) \rightarrow \min_{\theta}$$

$$\nabla_{\theta} L = \sum_{x, y \sim \text{Data}} (\nabla_{\theta} E(y, x, \theta) + \nabla_{\theta} \log Z(x, \theta))$$

$$\nabla_{\theta} \log Z(x, \theta) = -\mathbb{E}_{p(y|x, \theta)} \nabla_{\theta} E(y, x, \theta) \quad \text{- estimated using Gibbs sampling}$$

But we have chain structure between y_i , not an arbitrary connectivity graph. So we have effective sampling strategy:

$$y \sim p(y|x, \theta), \quad y_T \sim \exp(-\varphi_T(y_T, x, \theta)) \cdot \frac{1}{C_{\text{norm}}} \quad \text{1-dim distributions}$$
$$y_{T-1} \sim \exp(-\varphi_{T-1}(y_{T-1}, x, \theta) - \psi_{T-1}(y_{T-1}, y_T, x, \theta)) \cdot \frac{1}{C_{\text{norm}}}$$

etc. until y_1

Effective computation of z :

$$z(x, \theta) = \sum_y \exp(-E(y, x, \theta)) =$$
$$= \sum_{y_1} \underbrace{\exp(-\phi_1(y_1, x, \theta)) \sum_{y_2 \dots y_T} \exp(-\psi_1(y_1, y_2, x, \theta)) \cdot \exp(\dots)}_{z_1(y_1)}$$

unroll these summations and then reduce starting from y_T

$$z_T(y_T) = \exp(-\phi_T(y_T, x, \theta))$$

$$z_{T-1}(y_{T-1}) = \sum_{y_T} z_T(y_T) \cdot \exp(-\phi_{T-1} - \psi_{T-1}) =$$

$$= \exp(-\phi_{T-1}(y_{T-1}, x, \theta)) \sum_{y_T} z_T(y_T) \exp(-\psi_{T-1}(y_{T-1}, y_T, x, \theta))$$

... till z_1

$$z = \sum_{y_1} z_1(y_1)$$

Note, that z_i are functions and represented by a vector of K size, where K is the number of classes.

So, to compute z_i we need $O(K^2)$ operations.

And for z we have $O(T \cdot K^2)$ complexity.

For text segmentation we indeed need these pairwise potentials, because our entities are constructed of several tokens in a specific way.

Like $\langle \text{ent. start} \rangle \langle \text{ent. mid} \rangle \langle \text{ent. end} \rangle \langle \text{no ent.} \rangle$.

Such restrictions are imposed by ψ_i .

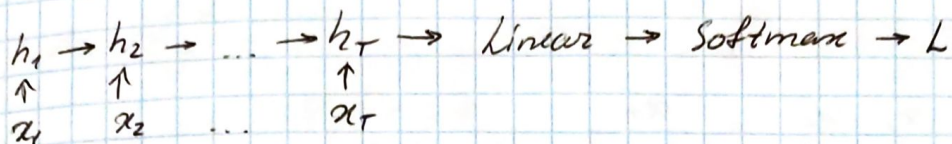
For ex.

$$\psi_i(\langle \text{ent. start} \rangle, \langle \text{no ent.} \rangle) \gg 1. \quad (=1000 \text{ for ex.})$$

Text recognition problem

Considered in "Recurrent NN" paragraph.

Text classification problem



But what are actually x_i in the computer?

Trivial option - one-hot encoding over all words in our vocabulary - too big vectors, too big matrices.

So, we need some tokenization; (for ex.)

all words $\rightarrow$ stemming $\rightarrow$ ~~remove~~ drop all least frequent words $\rightarrow$ $\langle \text{unk} \rangle$
 $\leftarrow$ remove different forms of words $\uparrow$
most frequent words

(~~drop~~ drop most frequent words because they ~~don't~~ don't impact on classification).

Now vectors aren't too big, but matrices still are.

So, we compute embeddings of fixed size for all tokens.

$$x_i \in \{1, \dots, V\} \rightarrow \text{emb}(x_i) \in \mathbb{R}^h$$

$\text{emb}(x_i)$ is taken from some pre-trained model.

Other ways for tokenization:

- nltk-tokenize (library)

- BPE

- word piece : if $\frac{\#(a, b)}{\#a \#b}$ is the highest, we merge ~~a~~ a and b $\leftarrow$ tokens

- seq piece (for hieroglyph languages)

Word embeddings

1. Word2vec

Our text: $x_1, x_2, \dots, x_T$

Consider sliding window. Embeddings should be close for close words.

$$\sum_t \sum_{c \in \mathcal{E}_t} \log p(\text{emb}(x_c) | \text{emb}(x_t)) \rightarrow \max_{\text{emb}}$$

↑
words in the vicinity
of current position t

$$p(\text{emb}(x_c) | \text{emb}(x_t)) = \frac{\exp(\text{score}(\text{emb}(x_t), \text{emb}(x_c)))}{\sum_{w=1} \exp(\text{score}(\text{emb}(x_t), \text{emb}(w)))}$$

sum over
all words in vocab.

$$\text{score}(u, v) = u^T v \quad \text{— in simplest case}$$

But since computing this big sum is expensive (and gives also numerical problems), we instead use negative sampling:

$$\sum_t \left(\sum_{c \in \mathcal{E}_t} \log \sigma(\text{score}(\text{emb}(x_t), \text{emb}(x_c))) + \sum_{h \in \mathcal{N}_{t,k}} \log \sigma(-\text{score}(\text{emb}(x_t), \text{emb}(x_h))) \right) \rightarrow \max_{\text{emb}}$$

words in a window around x_t , excluding x_t itself

negative samples (random words from the outside $\mathcal{E}_t$ vocabulary)

Usually $|\mathcal{N}_{t,k}| = |\mathcal{E}_t|$ (number of positive samples) (but seems not necessarily and $|\mathcal{N}_{t,k}|$ is defined by user)

This approach is general and can be used to construct embeddings for objects from different domains.

2. fasttext

It's a modification of Word2vec to account for rare words (or even non existing ones).

Consider a word '<where>' and all its 3-grams:

$$\{ \text{<wh, whe, her, ere, re>} \} = \text{3-gram}(\text{<where>})$$

↑
start and end symbols

And we make embeddings for these 3-grams (or n-grams in general) like in Word2vec.

$$\text{score}(x, y) = \left(\sum_{g \in \text{n-gram}(x)} \text{emb}(g) \right)^T \text{emb}(y)$$

in addition to embeddings for words from our vocab.

So, $\text{emb}(\cdot)$ is constructed for words and for n-grams.

CBOW: By set of context words we try to predict the target word

Skip-gram: By target word we try to predict surrounding context words.

Probably we may compute embeddings for rare words using sum of its n-gram embeddings.

More detailed and clear description:

~~Global vectors~~ $\xrightarrow{\text{let}}$ word of n symbols
 Let $\text{emb}(x)$ be embedding of word (or n-gram) taken from our embedding table.

$$\text{emb}^n(x) = \sum_{g \in \text{n-gram}(x)} \text{emb}(g)$$

$\text{score}(u, v) = u^T v$, where u and v are vectors.
 we probably can also use cosine similarity

Then we optimize the following:

$$\sum_t \left(\sum_{c \in E_t} \log \sigma(\text{score}(\text{emb}^n(x_t), \text{emb}(c))) + \sum_{h \in N_t} \log \sigma(-\text{score}(\text{emb}^n(x_t), \text{emb}(h))) \right) \rightarrow \text{max emb}$$

(comparing to Word2vec)
 Important thing here is that E_t includes x_t itself,

so we also learn sum of n-gram embeddings to represent the target word. After training we have

2 ways of obtaining word embedding:

1. By taking from emb. table.
2. By computing the sum of its n-grams embeddings.

The second way is the only choice for words, not presented in our emb. table (rare and non-existent ones).

$|E_+|$ and $|N_+|$ are hyperparameters and chosen by the user.

3. GloVe (global vectors)

Word2vec and fasttext don't account for the global statistics of our corpus: they make no difference if some two words coappear rarely or frequently.

number of times x_i and x_j appear together at a distance $\leq k$ (k is hyperparam)

$$\sum_{i,j=1}^V f(N_{ij}) (\text{emb}(x_i)^T \text{emb}(x_j) + b_i + b_j - \log N_{ij})^2 \rightarrow \min_{\text{emb}}$$

b_i, b_j are also learned

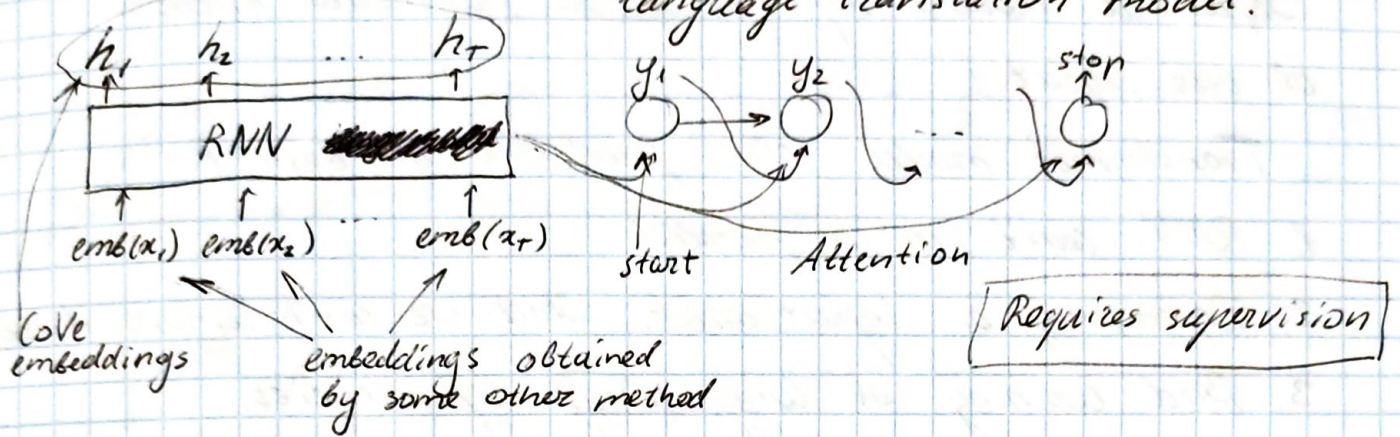
$f(N_{ij})$:

$N_{\max}$ hyperparameters

$$f(N_{ij}) = \begin{cases} (N_{ij}/N_{\max})^\alpha & N_{ij} < N_{\max} \\ 1 & N_{ij} \geq N_{\max} \end{cases}$$

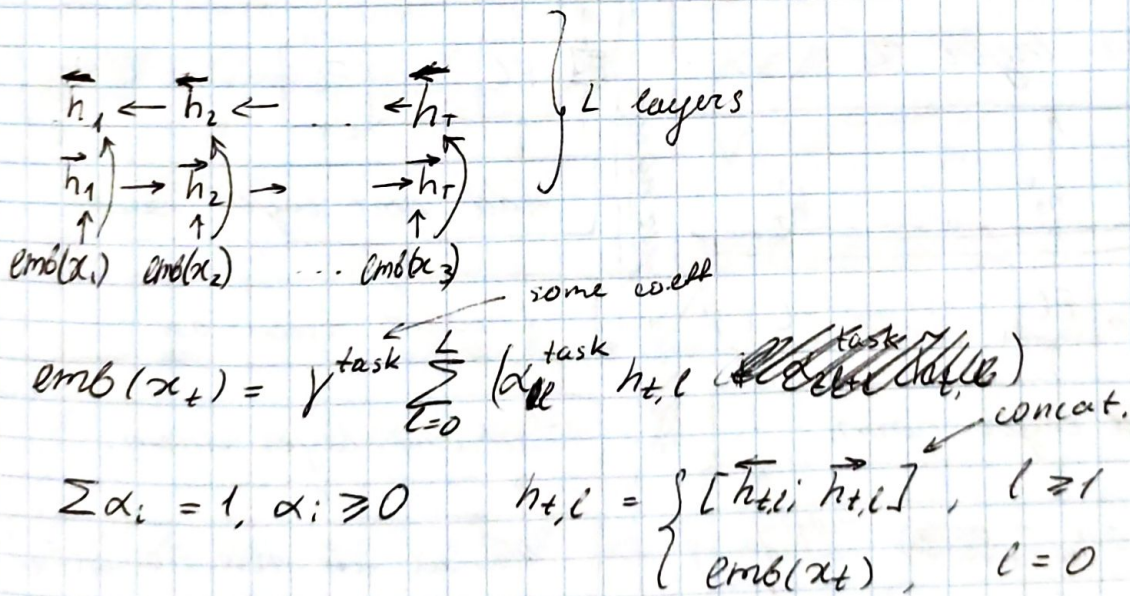
4. CoVe (contextualized vectors)

Above methods were local. Consider following machine language translation model:



CoVe embeddings represent context of words and are constructed for each text specifically. So, ~~one~~ the same word will have different CoVe embeddings in different text. RNN is originally bidirectional, so $h_1, h_2, \dots$ will have context of the following words too.

5. ELMo (embeddings from language model)

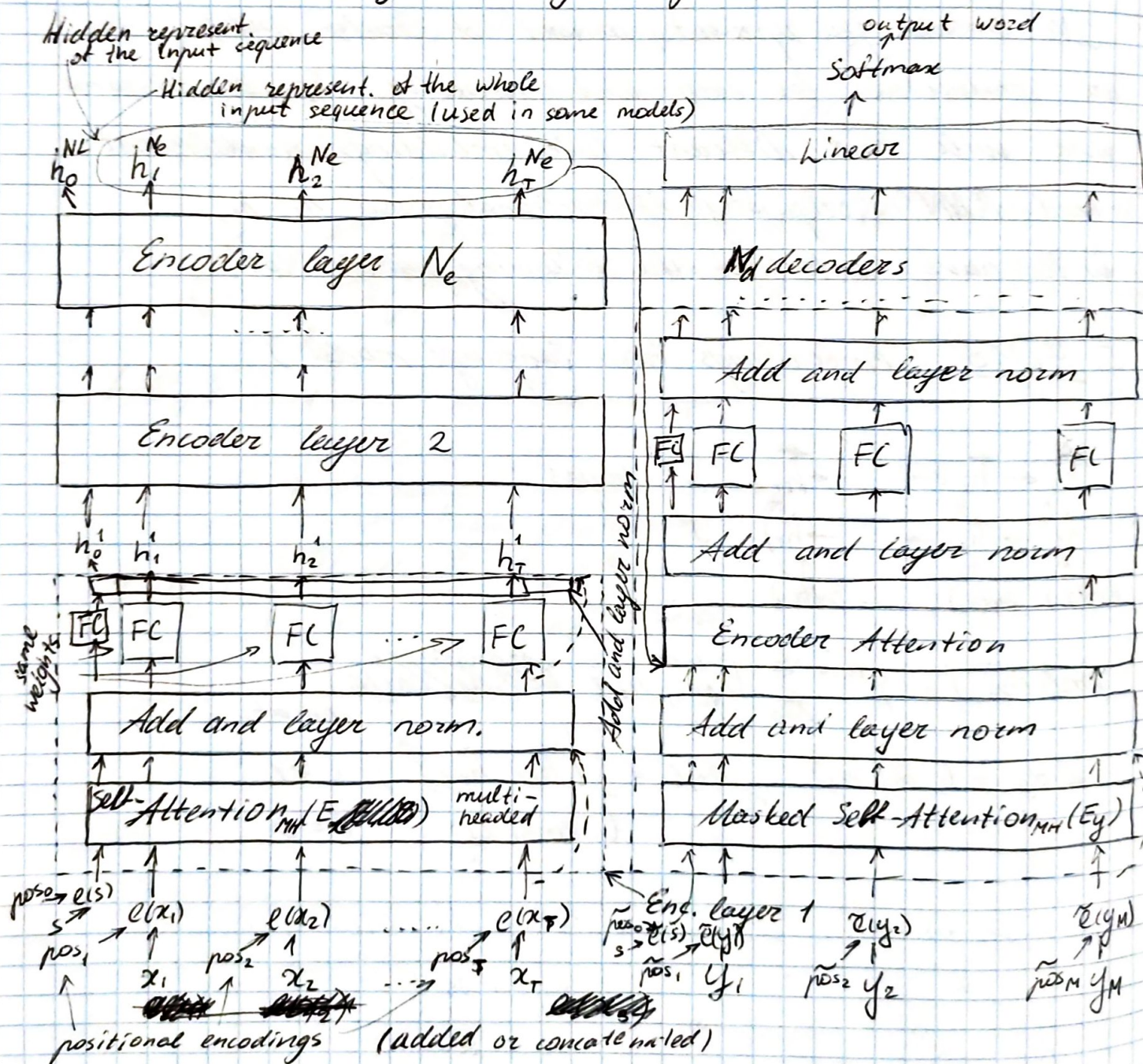


Transformer models

Sequential models depend on a particular order of our input.

Transformer model solves problems of Basic RNNs:

1. RNNs can't run in parallel.
2. Dependency on input order. (which we can't remove it needed)
3. Bad learning of long range dependencies.



Key idea: drop left to right connections.

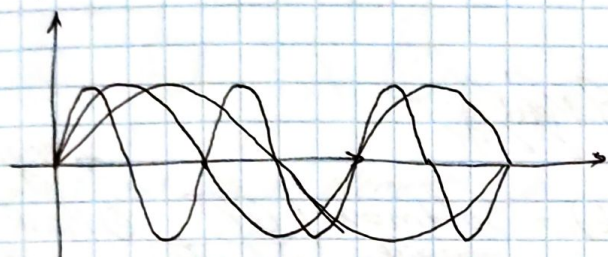
If we don't use positional embeddings, then our model will be ~~invariant~~ ^{invariant} to input order.

Classic way to compute positional encodings:

$$PE_{pos, 2i} = \sin\left(\frac{pos}{10000^{2i/d_{model}}} k_i\right)$$

$$PE_{pos, 2i+1} = \cos\left(\frac{pos}{10000^{2i/d_{model}}} k_i\right) \quad k_i = \frac{2i}{d_{model}}$$

$$PE(pos) \rightarrow \mathbb{R}^{d_{model}} \leftarrow \text{chosen by user}$$



Has following property:

$$\begin{aligned} PE_{pos+k, 2i} &= \sin\left(\frac{pos+k}{const}\right) = \sin\left(\frac{pos}{const}\right) \cos\left(\frac{k}{const}\right) + \cos\left(\frac{pos}{const}\right) \sin\left(\frac{k}{const}\right) \\ &= \text{Linear}(PE_{pos, 2i}, PE_{pos, 2i+1}) \end{aligned}$$

$$\begin{aligned} \text{Attention}(Q, K, V) &= \begin{matrix} Q \in \mathbb{R}^{T \times \dim(Q)} \\ K \in \mathbb{R}^{T \times \dim(Q)} \\ V \in \mathbb{R}^{T \times \dim(V)} \end{matrix} \\ &= \text{rowwise} \text{ Softmax}\left(\frac{QK^T}{\sqrt{\dim(Q)}}\right) \cdot V \end{aligned}$$

Q - matrix of queries for each input vector

K - matrix of ~~keys~~ keys for each input vector

V - matrix of values

Multi-headed Self-Attention:

$$A_i = \text{Self-Attention}_i(E) = \text{Attention}(E \cdot W_i^Q, E \cdot W_i^K, E \cdot W_i^V)$$

$$A_{MH} = \text{Self-Attention}_{MH}(E) = \text{concat}(\{ \text{Self-Attention}_i(E) \}_{i=1}^{N_H}) \cdot W^O$$

$$A_i \in \mathbb{R}^{T \times \dim(V)}$$

$$A_{MH} \in \mathbb{R}^{T \times (N_H \cdot \dim(V))}$$

$$W^O \in \mathbb{R}^{(\dim(V) \cdot N_H) \times \dim(E)}$$

$$E \in \mathbb{R}^{T \times \dim(E)}$$

$$W_i^Q, W_i^K \in \mathbb{R}^{\dim(E) \times \dim(Q)}$$

$$W_i^V \in \mathbb{R}^{\dim(E) \times \dim(V)}$$

W_i^Q, W_i^K, W_i^V are learnable matrices

$$\text{concat}(A_i) \in \mathbb{R}^{T \times (\dim(V) \cdot N_H)}$$

N_H - number of heads - hyperparam

Add another normalization layer:

In original paper

$$\dim(E) = 512$$

$$N_e = N_d = 6$$

$$\dim(V) = \dim(Q) = 64$$

$$N_H = 8$$

Masked Attention:

~~Masked Attention~~ $\frac{QK^T}{\sqrt{\dim(Q)}} \odot \text{Mask}$

$$\text{Masked Attention}(Q, K, V, \text{Mask}) = \underset{\text{rowwise}}{\text{Softmax}} \left(\frac{QK^T}{\sqrt{\dim(Q)}} \odot \text{Mask} \right) \cdot V$$

$+ \log(\text{Mask})$

~~Mask has 1 where we want~~ ^{in columns} consider $\log(0) = -\infty$
(it's only for formula)

At layer l of the decoder (when we ~~decode~~ generate word ~~yes~~ y_l). we have following Mask:

$$\text{Mask} = \begin{bmatrix} 1 & \dots & 1 & +0 & \dots & +0 \\ \vdots & & & & & \\ 1 & \dots & 1 & +0 & \dots & +0 \end{bmatrix} \quad \text{so we mask out not yet generated words}$$

$\underbrace{\quad\quad\quad}_l \quad \underbrace{\quad\quad\quad}_{T-l}$

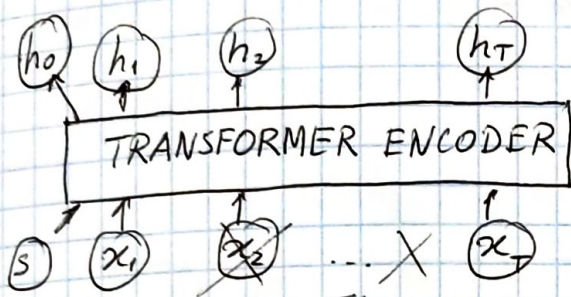
Encoder Attention:

In lecture we use hidden vectors from corresponding encoder layer, but in the paper we use vectors from the last layer of encoder. Probably, it depends on a particular implementation.

On the output we have prob. distribution over our vocabulary. (So, we run decoder stack for each output word in the sequence, paying attention to the previously generated words and encoding of the input sequence).

BERT (bidirectional encoder representation from transformer)

Trained in self-supervised manner.



Mask-out some elements and try to predict these elements in the output.

h_0 was trained ~~to~~ ~~receiving~~ on text classification task.

This is a basic model for using in specific text-related tasks.

Image transformer

Generative models. Variational autoencoder

Examples:

- generating handwriting given a text
- image transformer
- outlier detection

EM - algorithm (explanation. not from lectures)

Consider we have some discrete or continuous ~~multivariate~~ pdf (prob. distribution function), which takes x and z as input and depends on parameters θ (which in our practical case is defined by some NN_θ like $p(x, z | \theta) = \mathcal{N}(x | NN_\theta(z), I)$ (and on inference we just take $x = NN_\theta(z)$ as the most probable sample)).

$p(x, z | \theta) : \mathbb{R}^{n+m} \rightarrow \mathbb{R}[0, 1]$, ~~the~~
(or $(\mathbb{R} \cup \mathbb{N})^{n+m} \rightarrow \mathbb{R}[0, 1]$, meaning x and z may be discrete or continuous), $x \in (\mathbb{R} \cup \mathbb{N})^n$, $z \in (\mathbb{R} \cup \mathbb{N})^m$.

And ~~we want to find~~ for given observations we want to ~~find~~ find parameters θ , which maximize prob. of observing the data (likelihood):

$$\prod_{i=1}^D p(x_i, z_i | \theta) \rightarrow \max_{\theta}, \quad D = |\text{Data}|$$

But in our data we actually have only x_i and z_i are hidden. So, let's then optimize over all possible z_i :

$$\prod_{i=1}^D \underbrace{\int p(x_i, z_i | \theta) dz_i}_{= p(x_i | \theta)} \rightarrow \max_{\theta}$$

(Note, that z_i are independent and x_i are independent, so we may factorize $p(\mathbf{x}, \mathbf{z} | \theta) = \prod_{i=1}^D p(x_i, z_i | \theta)$)

In fact, we maximize log of likelihood, which allows simpler solutions:

$$\sum_i \log \int p(x_i, z_i | \theta) dz_i \rightarrow \max_{\theta}$$

However here we have summation under log in case of discrete distr. of z_i , or, even worse, integral over continuous z_i , which is probably intractable. And in this way we're unable to effectively do the optimization.

But we can transform this expression for each z_i :

$$\log \int p(x_i, z_i | \theta) dz_i = \log \int q_i(z_i) \frac{p(x_i, z_i | \theta)}{q_i(z_i)} dz_i \geq$$

| using Jensen's inequality | $\geq \int q_i(z_i) \log \frac{p(x_i, z_i | \theta)}{q_i(z_i)} dz_i \stackrel{\text{def}}{=} F_i(q_i, \theta)$

$\leftarrow$ ELBO

for arbitrary distribution $q_i(z_i)$.

Then all what we do is iteratively optimize

$$\sum_i F(q_i, \theta) \rightarrow \max_{\substack{q_i, \theta \\ i=1, D}} , \text{ alternating between } q_i \text{ and } \theta.$$

So, on k -th step we have:

E-step:

$$\sum_i F(q_i^k, \theta^{k-1}) \rightarrow \max_{q_i^k, i=1, D} , q_i^k = \operatorname{argmax}_{q_i} F(q_i, \theta^{k-1})$$

M-step:

$$\sum_i F(q_i^k, \theta^k) \rightarrow \max_{\theta^k} , \theta^k = \operatorname{argmax}_{\theta} F(q_i^k, \theta)$$

For E-step we can transform $F(q_i, \theta)$:

$$F(q_i, \theta) = \int q_i(z_i) \log \frac{p(x_i, z_i | \theta)}{q_i(z_i)} dz_i =$$

$$= \int q_i(z_i) \log \frac{p(z_i | x_i, \theta) p(x_i | \theta)}{q_i(z_i)} dz_i =$$

$$= \underbrace{\int q_i(z_i) \log p(x_i, \theta) dz_i}_{= \log p(x_i, \theta) \cdot 1 = \text{const w.r.t. } q_i} - \underbrace{\int q_i(z_i) \log \frac{q_i(z_i)}{p(z_i|x_i, \theta)} dz_i}_{KL(q_i(z_i) \parallel p(z_i|x_i, \theta))}$$

So, we minimize KL (Kullback-Leibler) divergence between $q_i(z_i)$ and $p(z_i|x_i, \theta)$.

For discrete z_i we can just take $q_i(z_i) = p(z_i|x_i, \theta)$, for continuous z_i $q_i(z_i)$ ^{there} will be some parametrized family of distributions $q_i(z_i|\lambda)$, where we optimize λ . (Using SGD, for example).

$p(z_i|x_i, \theta)$ is computed using Bayesian rule:

$$\cancel{p(z_i|x_i, \theta)} p(z_i|x_i, \theta) = \frac{p(x_i|z_i, \theta) \cdot p(z_i|\theta)}{\int \sum_{\mathbf{z}} p(x_i|z_i, \theta) p(z_i|\theta) dz_i} =$$

$$= \frac{p(x_i, z_i|\theta)}{\int \sum_{\mathbf{z}} p(x_i, z_i|\theta) dz_i}$$

(usually $p(x_i|z_i, \theta)$ and $p(z_i|\theta)$ are defined by our model and that's how we actually compute $p(z_i|x_i|\theta)$).

For M-step we also can transform $F(q_i, \theta)$:

$$F(q_i, \theta) = \int q_i(z_i) \log \frac{p(x_i, z_i|\theta)}{q_i(z_i)} dz_i =$$

$$= \underbrace{\int q_i(z_i) \log p(x_i, z_i|\theta) dz_i}_{= \mathbb{E}_{q_i(z_i)} \log p(x_i, z_i|\theta)} - \underbrace{\int q_i(z_i) \log q_i(z_i) dz_i}_{\text{const w.r.t. } \theta}$$

So, we maximize expectation over $q_i(z_i)$:

$$\mathbb{E}_{q_i(z_i)} \log p(x_i, z_i|\theta) \rightarrow \max_{\theta}$$

by directly computing the expectation (if possible) or by computing its stochastic estimate.

Log-derivative trick: (REINFORCE)

Suppose we ~~have~~ need to find grad w.r.t. to ~~the~~ expectation:

$$\nabla_{\lambda} \mathbb{E}_{q(z|\lambda)} f(x, z, \theta, \lambda) \quad \text{for some pdf } q(z|\lambda) \text{ and some function } f(x, z, \theta, \lambda).$$

We have $\nabla_{\lambda} \mathbb{E}_{q(z|\lambda)} f(x, z, \theta, \lambda) = \int \nabla_{\lambda} q(z|\lambda) f(x, z, \theta, \lambda) dz + \int q(z, \lambda) \nabla_{\lambda} f(x, z, \theta, \lambda) dz =$

the trick $\rightarrow$

$$= \left| \begin{array}{l} \nabla_{\lambda} \log q(z|\lambda) = \frac{\nabla_{\lambda} q(z|\lambda)}{q(z|\lambda)} \\ \nabla_{\lambda} q(z|\lambda) = q(z|\lambda) \cdot \nabla_{\lambda} \log q(z|\lambda) \end{array} \right| =$$

can now be estimated stochastically

$$= \mathbb{E}_{q(z|\lambda)} (\nabla_{\lambda} \log q(z|\lambda) \cdot f(x, z, \theta, \lambda) + \nabla_{\lambda} f(x, z, \theta, \lambda)).$$

So, it's the way to propagate ∇ through expectation, depending on the parameters being differentiated.

Reparametrization trick:

Suppose we again need to find $\nabla_{\lambda} \mathbb{E}_{q(z|\lambda)} f(x, z, \theta, \lambda)$.

If ~~we can express~~ z can be expressed in a way

$z = g(z_0, \lambda)$, where $z_0 \sim q_0(z_0)$, $g(z_0, \lambda)$ is ^{some function} differentiable w.r.t. λ and $q_0(z_0)$ is independent from λ

preserving $z = g(z_0, \lambda) \sim q(z|\lambda)$, then

$$\begin{aligned} \nabla_{\lambda} \mathbb{E}_{q(z|\lambda)} f(x, z, \theta, \lambda) &= \nabla_{\lambda} \mathbb{E}_{q_0(z_0)} f(x, g(z_0, \lambda), \theta, \lambda) = \\ &= \mathbb{E}_{q_0(z_0)} \nabla_{\lambda} f(x, g(z_0, \lambda), \theta, \lambda). \end{aligned}$$

So, it's ~~another~~ alternative to log-derivative trick.

The main advantage is that ^{the variance of} stochastic estimate of this expectation may be orders of magnitude smaller, than of expectation, obtained by log-derivative trick.

Lecture 8

(DL)

Generative models,
VAE, VQVAE

We have

$\{x_1, x_2, \dots, x_n\}$ - dataset

~~x~~ $x \sim p(x)$ ← assumption
↑
distribution

We want to sample from this distribution. - ~~so~~ so we generate new objects, similar to ones from the dataset

Examples: • generate hand-written text.

• pronounce some text ~~with~~ ^{with} someone's ~~voice~~ ^{voice}
~~be in a different way~~ and in a different way

Properties:

- 1) unsupervised
- 2) conditional (on text, or other data)

How to solve:

Introduce z - hidden embedding for an object.

$$z \rightarrow NN_{\theta} \rightarrow x$$

↑
neural network
depending on θ

$$z \sim p(z)$$

$$x \sim p(x|z) = p(x|NN_{\theta}(z))$$

Simplest case:

$$z \in \mathbb{R}^d, x \in \mathbb{R}^D$$

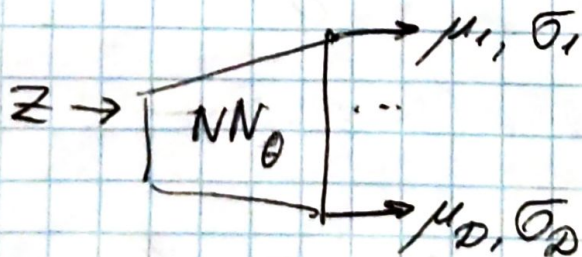
$$p(z) = \mathcal{N}(z | 0, I)$$

$$p(x|z) = \mathcal{N}(x | \mu(x, \theta), \text{diag}(\text{var}_1(x, \theta), \dots, \text{var}_D(x, \theta)))$$

← cov. matrix
↑
output from NN

$$\text{var}_i(x, \theta) = \log(1 + \exp(\sigma_i(x, \theta)))$$

so > 0



If D is small $\rightarrow$ full cov. matrix

$$p(x|z) = \mathcal{N}(x | \mu(x, \theta), \Sigma(x, \theta))$$

(note that Σ must be symmetric and positively-defined)

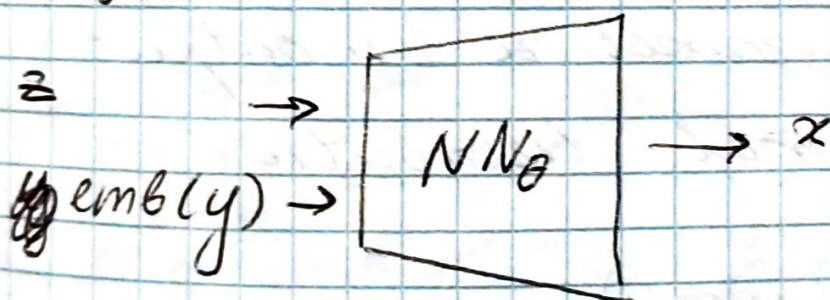
$$\Sigma(x, \theta) = L(x, \theta) L(x, \theta)^T, \text{ where}$$

$$L = \begin{bmatrix} \Delta & 0 \end{bmatrix} - \text{arbitrary low triangular matrix}$$

Conditional generative model:

$$p(x|z, y)$$

$$p(z|y)$$



Case of x is not $\mathbb{R}$

For ex. x - MNIST image

$$x \in \{0, 1\}^D$$

$$p(x|z) = \prod_{i=1}^D \text{Bern}(x_i | p_i(x, \theta))$$

$$\sigma(\tilde{p}_i(x, \theta))$$

output from NN

If not binarize:

$$x \in [0, 1]^D$$

If we don't binarize:

$$x \in [0, 1]^D$$
$$p(x|z) = \prod_{i=1}^D \text{Beta}(x_i; \alpha_i(z, \theta), \beta_i(z, \theta))$$

Beta distribution
 $\alpha_i, \beta_i > 0$
use soft+ transform
($\log(1 + \exp(x))$)

But we would like to introduce distribution over arbitrary domain.

So we reparametrize an output from a normal distribution.

$$u \sim \mathcal{N}(u | \mu, \sigma^2)$$
$$x = \sigma(u) \in (0, 1)$$

just some sigmoid

generative

$$p(x) = \text{Logit-Normal}(x | \mu, \sigma^2)$$

In case we need a distribution over a simplex:

$$x \in \mathbb{R}^D, \quad x_i > 0, \quad \sum_i x_i = 1$$

~~we~~ then

$$u \sim N(u | \mu, \Sigma), \quad x = \text{softmax}(u)$$

and so on.

How do we learn?

$\{x_1, \dots, x_N\}$ - objects z_i - embeddings

$$p(X, Z | \theta) = \prod_{i=1}^N p(x_i | z_i, \theta) p(z_i)$$

NN params

may also depend on θ

Options for learning:

① Full Likelihood ^{Maxim.} ~~optimization~~

$$\log p(X, Z | \theta) \rightarrow \max_{\theta, Z}$$

② Incomplete Likelihood Maxim

$$\log p(X | \theta) = \log \int p(X, Z | \theta) dZ \rightarrow \max_{\theta}$$

is not tractable

So we ~~can~~ introduce lower bound like in EM algorithm.

ELBO:

$$\log p(X|\theta) \geq \mathbb{E}_{q(z)} \log \frac{p(X, Z|\theta)}{q(z)} \quad \leftarrow \text{arbitrary distribution}$$

$$\log p(X|\theta) \geq \mathbb{E}_{q(z)} \log \frac{p(X, Z|\theta)}{q(z)} \rightarrow \max_{\theta, q(z)} \quad \text{for arbitrary distrib. } q \uparrow \text{ is tractable}$$

We say that

$$q(z) = q(z|\lambda) \quad \text{for ex } q \text{ is } \mathcal{N} \text{ and } \lambda \text{ is } \mu, \sigma^2$$

$$\text{ELBO}(\theta, \lambda) := \mathbb{E}_{q(z|\lambda)} \log \frac{p(X, Z|\theta)}{q(z|\lambda)} \rightarrow \max_{\theta, \lambda}$$

$$\text{if } q(z) = p(z|X, \theta)$$

$$\text{then } \geq \rightarrow =$$

How to compute stochastic gradient?

$$\nabla_{\theta} \text{ELBO}(\theta, \lambda) = \nabla_{\theta} \mathbb{E}_{q(z|\lambda)} \log p(X, Z|\theta) = \mathbb{E}_{q(z|\lambda)} \nabla_{\theta} \log p(X, Z|\theta) \approx$$

choose by ourselves,
so can easily sample
from it

Monte-Carlo samples
from $q(z|\lambda)$

$$\approx \frac{1}{M} \sum_{j=1}^M \nabla_{\theta} \log p(X, z^j|\theta)$$

$$\begin{aligned} \nabla_{\lambda} \text{ELBO}(\theta, \lambda) &= \nabla_{\lambda} \mathbb{E}_{q(z|\lambda)} \log \frac{p(x, z|\theta)}{q(z|\lambda)} = \\ &= \left\{ \begin{array}{l} \text{reparam. trick} \\ \mathbf{z} \sim q(\mathbf{z}|\lambda) \Leftrightarrow \mathbf{e} \sim q_0(\mathbf{e}) \\ \mathbf{z} = f(\mathbf{e}, \lambda) \end{array} \right\} \end{aligned}$$

differentiable $\rightarrow$

$$\begin{aligned} &= \nabla_{\lambda} \mathbb{E}_{q_0(\mathbf{e})} \log \frac{p(x, f(\mathbf{e}, \lambda)|\theta)}{q(f(\mathbf{e}, \lambda)|\lambda)} = \\ &= \mathbb{E}_{q_0(\mathbf{e})} \nabla_{\lambda} \dots \end{aligned}$$

Which distributions are reparametrizable?

$$\mathbf{z} \sim \mathcal{N}(\mathbf{z}|\mu, \Sigma) \Leftrightarrow$$

$$\Leftrightarrow \mathbf{e} \sim \mathcal{N}(\mathbf{e}|0, \mathbf{I})$$

$$\mathbf{z} = \mathbf{L}\mathbf{e} + \mu, \quad \Sigma = \mathbf{L}\mathbf{L}^T$$

$$p(x, z | \theta) = \prod_{i=1}^N p(x_i | z_i, \theta) p(z_i)$$

$$q(z | \lambda) \approx p(z | x, \theta) = \prod_{i=1}^N p(z_i | x_i, \theta)$$

in ideal situation

$$q(z | \lambda) = \prod_{i=1}^N q(z_i | \lambda)$$

$$\log p(x | \theta) \geq \mathbb{E}_{q(z | \lambda)} \log \frac{p(x, z | \theta)}{q(z | \lambda)} \quad \textcircled{=}$$

$$\textcircled{=} \sum_{i=1}^N \mathbb{E}_{q(z | \lambda)} \log \frac{p(x_i | z_i, \theta) p(z_i)}{q(z_i | \lambda)}$$

$$= \sum_{i=1}^N \left[\mathbb{E}_{q(z_i | \lambda)} \log p(x_i | z_i, \theta) + \right.$$

$$\left. + \mathbb{E}_{q(z_i | \lambda)} \log \frac{p(z_i)}{q(z_i | \lambda)} \right] \xrightarrow{\theta, \lambda} \text{reconstruction term}$$

$$\uparrow -KL(q(z_i | \lambda) \| p(z_i)) \text{ "given a particular hidden var } z_i \text{ how well I describe my observed data } x_i?"$$

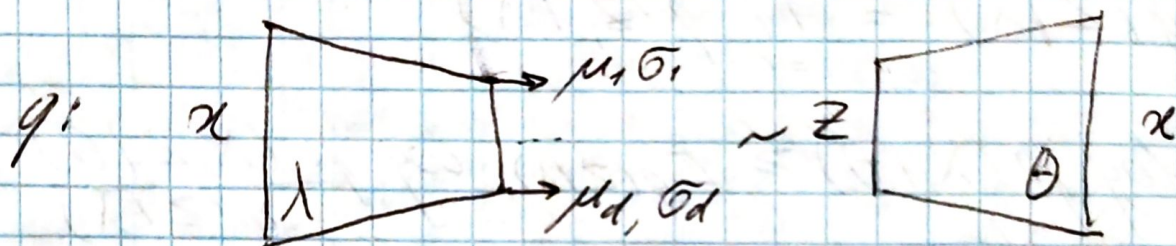
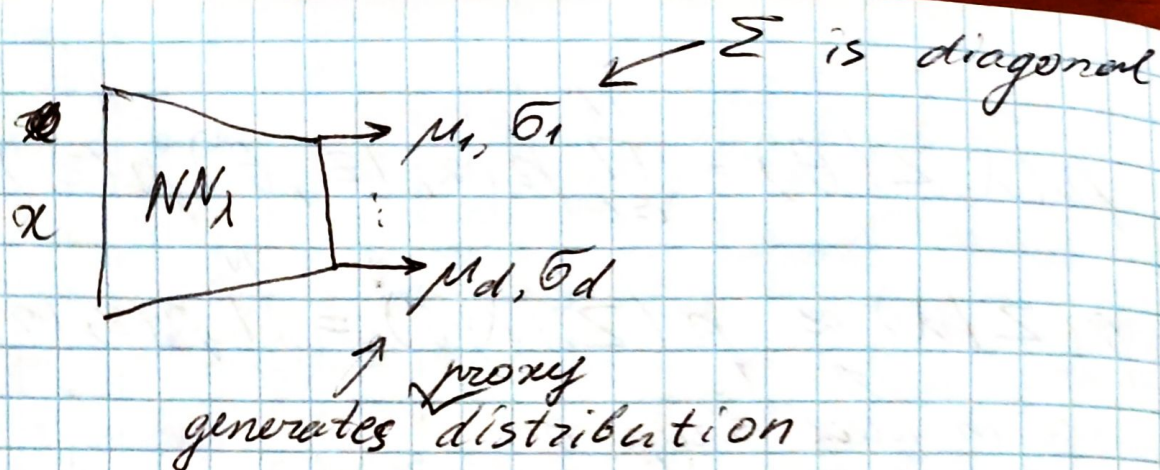
regularizer (minimize KL diverg.)

Approaches: introducing unconvenient cause params for each object

$$\textcircled{1} q(z_i | \lambda) = \mathcal{N}(z_i | \mu_i, \Sigma_i) \quad \forall i = 1, \dots, N$$

$$\textcircled{2} q(z_i | \lambda) = q(z_i | x_i, \lambda) = \mathcal{N}(z_i | \mu(x_i, \lambda), \Sigma(x_i, \lambda))$$

$\mu(x_i, \lambda)$ $\rightarrow$ output from NN
 $\Sigma(x_i, \lambda)$



$$p_d(x|z, y)$$

$$p_\theta(z|y) \quad ?$$

$$q_\lambda(z|x, y)$$

How to ~~score~~ score generated results?

$$p(x_{\text{test}}|\theta) = ?$$

$$\log p(x|\theta) \geq \mathbb{E}_{q(z|x, \theta)} \log \frac{p(x, z|\theta)}{q(z|x, \theta)} +$$

$$+ \text{KL}(q(z|x, \theta) \parallel p(z|x, \theta))$$

$\leftarrow$ we don't know

$$\textcircled{1} p(x) = \int p(x|z)p(z)dz \approx$$

$$\approx \frac{1}{M} \sum_{i=1}^M p(x, z^i), \quad z^i \sim p(z)$$

better:

joint distrib.

$$\textcircled{2} p(X) = \int p(X, Z) dZ =$$

$$= \int \frac{p(X, Z)}{q(Z|X, \theta)} q(Z|X, \theta) dZ \approx$$

$$\approx \frac{1}{M} \sum_{j=1}^M \frac{p(X, Z^j)}{q(Z^j|X, \theta)} \quad Z^j \sim q(Z|X, \theta)$$

$\textcircled{3}$ Importance weight autoencoder (IWAE)

$$\log p(X) = \log \left(\frac{1}{K} \sum_{j=1}^K p(X) \right) =$$

$$= \log \left(\frac{1}{K} \sum_{j=1}^K \int p(X, Z_j) dZ_j \right) =$$

$$= \log \left(\frac{1}{K} \sum_{j=1}^K \mathbb{E}_{q(Z_j|X)} \frac{p(X, Z_j)}{q(Z_j|X)} \right) =$$

$$= \log \mathbb{E}_{q(Z_1|X) \dots q(Z_K|X)} \frac{1}{K} \sum_{j=1}^K \frac{p(X, Z_j)}{q(Z_j|X)} \geq$$

$$\geq \text{[Jensen's inequality]} \geq \mathbb{E}_{q(Z_1|X) \dots q(Z_K|X)} \log \left(\frac{1}{K} \sum_{j=1}^K \frac{p(X, Z_j)}{q(Z_j|X)} \right) = \text{IWAE}_K(X)$$

$$\log \left(\frac{1}{K} \sum_{j=1}^K \frac{p(X, Z_j)}{q(Z_j|X)} \right) = \text{IWAE}_K(X)$$

~~IWAE pro~~

JWAE properties:

① $JWAE_1 = ELBO$

② $JWAE_{m+1} \geq JWAE_m$

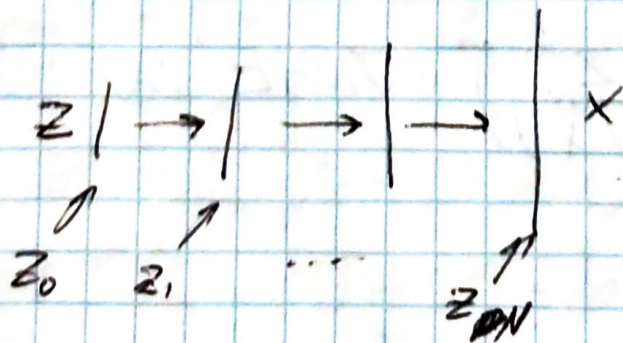
③ $JWAE_{\infty} = \log p(x)$

We choose K and with it
 $JWAE$ ^{letter} estimates likelihood for
testing object.

Improvement of VAE is
Hierarchical VAE

prior distribution $p(z|\theta) = p(z_0|\theta)p(z_1|z_0,\theta)\dots$
 $\dots p(z_N|z_0,\dots,z_{N-1},\theta)$

we generate z for all intermediate
partial resolutions



$$p(x|Z, \theta) = p(x|Z_N, \theta)$$

$$q(z|\lambda, \lambda) = q(z_0|x, \lambda) q(z_1|z_0, x, \lambda) \dots$$

$$\dots q(z_N | z_0, \dots, z_{N-1}, \lambda, \lambda) \phi$$

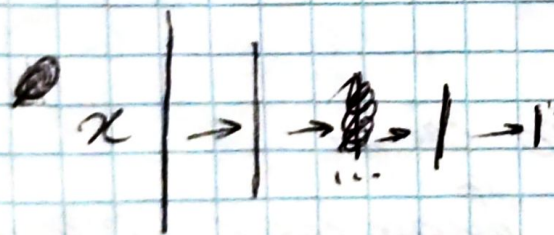
$$\log p(x) \geq E_{q(z|x, \lambda)} \log \frac{p(x, z | \theta)}{q(z | x, \lambda)} =$$

$$= E_{q(z_{1:n}, \lambda)} \log p(x | z_n, \theta) - KL(q(z_0 | x, \lambda) \| p(z_0 | \theta)) - KL(q(z_n | z_0, z_{n-1}, x, \lambda) \| p(z_n | z_0, z_{n-1}, \theta))$$

$$\rightarrow \max_{\theta, \lambda}$$

↑
regularizers

reconstruction
terms



$$\hat{z}_0 \sim N(z_0, 10, \tau)$$

Generative Adversarial Networks

Simple GAN:

$$x_1, \dots, x_N, x_i \sim p(x)$$

Want to sample $\hat{x}_1, \dots$ from similar distr.

$q \approx p(x)$ - hard to estimate for high dim. space.

So, instead we replace it with binary classification problem:

$$\hat{x} = g(z | \theta_g) = G$$

$z \sim p(z)$ - some primary distribution (say, $N(0, I)$)

Then we try to discriminate true image x from generated one $\hat{x}$.

$D = d(x, G) \rightarrow \mathbb{R}[0, 1]$ - prob. of x being true image.

let $p(x)$ - true data distr.

$q(x)$ - generated data distr. $x \sim q(x) = G(x)$

$p(z)$ - some standard distr. on z . $z \sim p(z)$

log-likelihood:

$$V(D, G) = \mathbb{E}_{x \sim p(x)} \log D(x) + \mathbb{E}_{z \sim p(z)} \log (1 - D(G(z)))$$

- objective for D to maximize. and for G to minimize.

Goal: $V(D, G) \rightarrow \min_G \max_D$

For fixed G :

$$V(D, G) \rightarrow \max_D$$

For fixed D :

$$\mathbb{E}_{z \sim p(z)} \log (1 - D(G(z))) \rightarrow \min_G$$

And repeat. But do it synchronously (each iteration do few SGD steps for D and do a step for G).

Cause there is asymmetry in D and G (G follows D),
so we do k SGD steps for D per one step for G .

After training, we may sample or interpolate in latent space to produce new objects.

Optimal D^* given fixed G :

$$D^*(x | G) = \frac{p(x)}{p(x) + q(x)} \quad \text{if } D \text{ is flexible enough (can output arbitrary value for each } x)$$

Given optimal D^* , optimal G^* will yield $q(x) = p(x)$.

But in practice neither we train optimal discriminator nor it's flexible enough.

And we don't fine tune G under optimal D .

~~Our~~ Our task is to find a saddle point ~~of~~ of V , ~~which~~ which is hard and unstable.

Problems:

On early iterations G generates poor takes.

D easily discriminates G , making $D(G(z)) \approx 0 \Rightarrow$

$$\nabla_{\theta_g} \log(1 - D(G(z))) \approx 0.$$

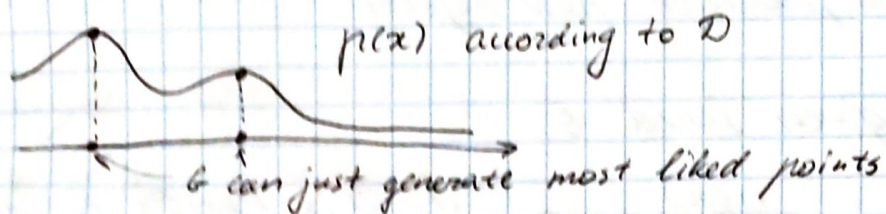
One solution: maximize

$$\mathbb{E}_{z \sim p(z)} \log D(G(z)) \rightarrow \max_G \Rightarrow \nabla_{\theta_g} \log D(G(z)) \neq 0$$

Another:

- occasionally flip labels when training D , so it won't be very confident.
- use soft labels instead of 1, and 0.

Mode collapse:



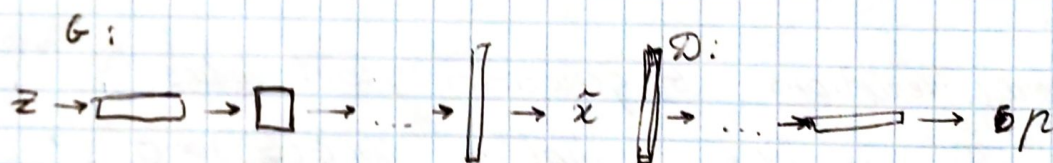
Solutions:

- add stochasticity to G (dropout, random noise in layers)
- add penalty for close $G(z_i)$ and $G(z_j)$ for $z_i, z_j \in \text{mini-batch}$. So, we want to maximize

$$\sum_{i \neq j} \|G(z_i) - G(z_j)\|_2^2$$

So, $V(G) - \lambda \sum_{i \neq j} \|G(z_i) - G(z_j)\|_2^2 \rightarrow \min$

Deep convolutional GAN (DCGAN)



Features:

- strided and fractional strided convolutions instead of pooling (cause pooling makes loss of spatial information, plus strided convolution is like "learnable pooling")
- LeakyReLU in the discriminator, to always have non-trivial gradient for G .
- batch normalization
- fully convolutional (without FC)
- ReLU in generator and tanh only on output.

We can do arithmetics ~~in~~ ⁱⁿ latent space.

Wasserstein GAN

$$x = g(z | \theta_g), \quad x \sim q(x | \theta_g)$$

We want $P(p(x), q(x | \theta_g)) \rightarrow \min_{\theta_g}$ for some distance P between distributions.

Popular distances:

Total variation (TV):

$$\delta(p, q) = \sup_{A \in \Sigma} |p(A) - q(A)|, \text{ where } \Sigma \text{ is all measurable subsets of } X$$

Kullback-Leibler (KL) divergence:

$$KL(p \parallel q) = \int p(x) \log \frac{p(x)}{q(x)} dx$$

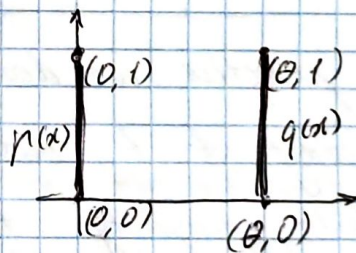
Jensen-Shannon (JS) divergence:

$$JS(p, q) = KL(p \parallel a) + KL(q \parallel a), \quad a = \frac{p+q}{2}$$

- symmetric

If we consider such distributions, the above metrics will be

discontinuous. (For non intersecting distributions they won't say how far they are).



Earth-Mover (EM) distance or Wasserstein-1:

$$W(p, q) = \inf_{\gamma \in \Pi(p, q)} \int \int \|x - y\| \gamma(x, y) dx dy$$

Π is a set of distributions (densities) $\gamma \in \Pi(p, q) : \mathbb{R}^2 \rightarrow \mathbb{R}[0, 1]$, if $p, q : \mathbb{R}^D \rightarrow \mathbb{R}[0, 1]$

that $\gamma \in \Pi(p, q)$, $\int \gamma(x, y) dy = p(x)$
then the following $\int \gamma(x, y) dx = q(y) \equiv q(x)$
must be satisfied:

For example for $p, q = \text{Unit}\{1, 2, 3\}$, possible $\gamma \in \Pi(p, q)$:

Y.

3	1/9	1/9	1/9
2	1/9	1/9	1/9
1	1/9	1/9	1/9

1 2 3 → p

or

3	0	0	1/3
2	0	1/3	0
1	1/3	0	0

1 2 3 → p

and so on
iso γ is not just joint
distr. of p and q

Properties of EM distance:

- if ~~EM~~ $q \xrightarrow{EM} p \Rightarrow q \xrightarrow{TV/KL/\#S} p$
- $W(p, q) \rightarrow 0 \Leftrightarrow q \xrightarrow{EM} p$
- if q_θ continuous in $\theta \Rightarrow W(p, q_\theta)$ continuous in θ . And if q_θ is diff. w.r.t. θ , then $W(p, q_\theta)$ is diff. w.r.t. θ .

How to calculate (estimate) $W(p, q)$?

Kantorovich - Rubinstein duality:

$$W(p, q_\theta) = \sup_{\|f\|_L \leq 1} (\mathbb{E}_{x \sim p}[f(x)] - \mathbb{E}_{x \sim q_\theta}[f(x)])$$

$\|f\|_L \leq 1$ - Lipschitz condition (property) ← doesn't matter, cause it's just multiplier for the supremum

($f'(x) \leq 1$ or $\|\nabla f(x)\|_2 \leq 1$ for diff. f)

(Lipschitz property: $f(x) - f(x') \leq K \cdot \|x - x'\|_2$).

In practice we don't care of K in Lipschitz property of f , but require it to satisfy some Lipschitz property.

$f(x|\theta_c)$ - critic, is a neural network.

with constraint on weights on each layer (constrain them to be not higher than some threshold)

WGAN Training:

Init θ_g, θ_c

until θ_g convergence do:

for $t = 0, \dots, n_{critic}$ do

$\{x^i\}_{i=1}^m \sim \text{Data}$

$\{z^i\}_{i=1}^m \sim p(z)$

$$g_c = \nabla_{\theta_c} \frac{1}{m} \sum_{i=1}^m (f(x^i|\theta_c) - f(G(z^i|\theta_g)|\theta_c))$$

$$\begin{aligned} \theta_c &\leftarrow \theta_c + \alpha_c g_c \quad \begin{array}{l} \text{or other opt. step} \\ \text{maximize } W \text{ w.r.t. } \theta_c \end{array} \\ \theta_c &\leftarrow \text{clip}(\theta_c, -c, c) \quad \text{some threshold (hyperparam)} \\ \{z^i\} &\sim p(z) \\ g_g &= \nabla_{\theta_g} \frac{1}{n} \sum_{i=1}^n t(G(z^i | \theta_g) | \theta_c) \\ \theta_g &\leftarrow \theta_g + \alpha_g g_g \quad \text{minimize } W \text{ w.r.t. } \theta_g \end{aligned}$$

Benefits of WGAN:

- doesn't have mode collapse problem.

Also, instead of clipping θ_c , we can add a regularization term $\lambda (\|\nabla_x t(x | \theta_c) - 1\|_2 - 1)^2$, because Lipschitz property is $\|\nabla_x t(x | \theta_c)\|_2 \leq 1$. (such approach works better)

Semi-supervised GAN

Instead of training a discriminator, which outputs just one prob. of x being true, we train a classifier, outputting (~~prob~~ $p(y=1|x), \dots, p(y=C|x), p(x \text{ is fake} | x)$). This way we get better training.

SGAN ~~works~~ achieves higher accuracy (on smaller datasets) than usual unsupervised GANs with binary-classification discriminator.

Conditional GAN (pix2pix)

$$z \sim p(z)$$

x - source version of object (say, some drawing)

y - target version of object (say, a photo, containing drawn object from x)

$\hat{y} = G(x, z)$ - reconstruction of x .

Objective:

$$L_{\text{GAN}}(G, D) = \mathbb{E}_{x, y \sim p(x, y)} \log D(x, y) + \mathbb{E}_{x, z \sim p(x), p(z)} \log(1 - D(x, G(x, z)))$$

~~$L_{\text{GAN}}(G, D)$~~

$$L_{\text{L1}}(G) = \mathbb{E}_{x, y, z \sim p(x, y), p(z)} \|y - G(x, z)\|_1 \quad \text{- reconstruction loss}$$

$$G^* = \arg \min_G \max_D (L_{\text{GAN}}(G, D) + \lambda L_{\text{L1}})$$

If to optimize only L_{L1} the results won't be realistic.

Generator uses U-Net-like architecture.

Discriminator is applied to patches of images and final prob. is taken as average between all patches (it's like CNN with avg. pooling at the end).

(because there is no benefit of convolving whole image or use too big patches).

Drawback:

- requires construction of dataset of pairs (x, y) .

Cycle GAN

$x_1, \dots, x_N \in X, y_1, \dots, y_M \in Y$ - unpaired dataset.

~~L_{GAN}~~ We have now $G_{X \rightarrow Y}, G_{Y \rightarrow X}, D_X, D_Y$ -

- generators from X to Y and from Y to X and discriminators for these domains.

$$L_{GAN}(G_{xy}, D_y) = \mathbb{E}_{y \sim p(y)} \log D_y(y) + \mathbb{E}_{x \sim p(x)} \log(1 - D_y(G_{xy}(x)))$$

$$L_{GAN}(G_{yx}, D_x) = \mathbb{E}_{x \sim p(x)} \log D_x(x) + \mathbb{E}_{y \sim p(y)} \log(1 - D_x(G_{yx}(y)))$$

$$L_{cyc}(G_{xx}, G_{yy}) = \mathbb{E}_{x \sim p(x)} \|G_{yx}(G_{xx}(x)) - x\|_1 + \mathbb{E}_{y \sim p(y)} \|G_{xy}(G_{yy}(y)) - y\|_1$$

Objective:

$$G_{xx}^*, G_{yy}^* = \arg \min_{G_{xx}, G_{yy}} \max_{D_x, D_y} (L_{GAN}(G_{xy}, D_y) + L_{GAN}(G_{yx}, D_x) + L_{cyc}(G_{xx}, G_{yy}))$$

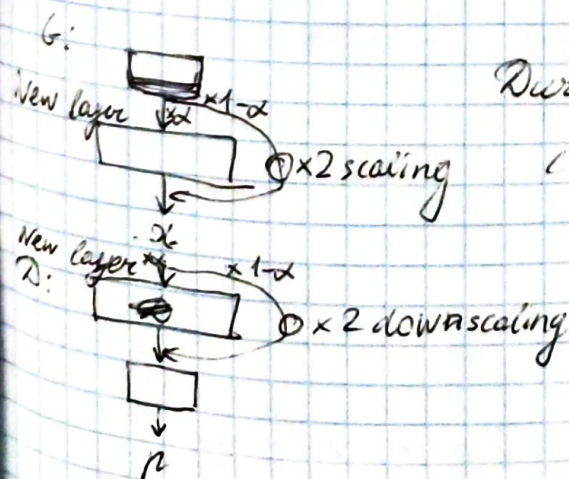
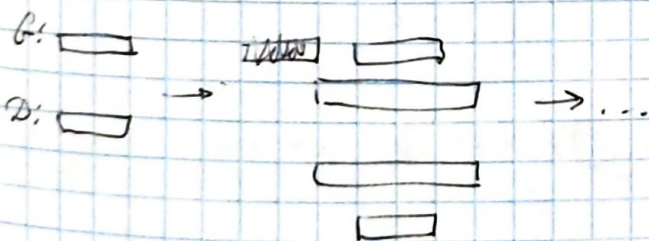
For style transfer to remove color change we introduce additional identity loss:

$$L_{identity} = \mathbb{E}_{x \sim p(x)} \|G_{yx}(x) - x\|_1 + \mathbb{E}_{y \sim p(y)} \|G_{xy}(y) - y\|_1$$

- so we ask to do nothing when source object is from target domain. (It removes shift in target space)

Progressive GAN

Start from training small images (2×2) and after each training add new layers, fixing previous ones. And do it until we reach needed ^{resolutions} quality



During training α gradually changes from 0 to 1.

Diffusion models

VAE: $p(z)$ ← prior distrib.
 generator: $p_\theta(x|z)$,
 to sample from $p_\theta(x|z)$ we train $q_\lambda(z|x) \approx p(z|x, \theta)$

Hierarchical VAE: $p(z_0), p_\theta(z_1|z_0) \dots p_\theta(z_N|z_{N-1})$
 $p_\theta(x|z_N)$

$$q(z|x) = q_\lambda(z_0|x) \dots q_\lambda(z_N|z_{N-1}, x)$$

Diffusion models:

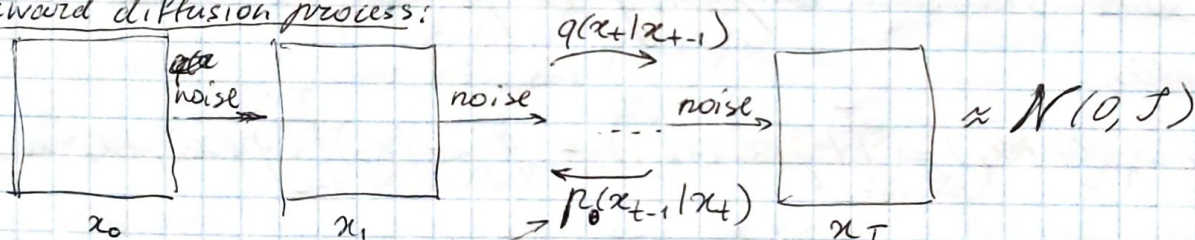
(we first introduce q)

x_0 - observed object (image)

$x_1, \dots, x_T$ - hidden variables

$$q(x_1, \dots, x_T | x_0) = q(x_1|x_0) q(x_2|x_1) \dots q(x_T|x_{T-1})$$

Forward diffusion process:



reverse process

needed to preserve the variance on each time step

$$q(x_t|x_{t-1}) = \mathcal{N}(x_t | \sqrt{1-\beta_t} x_{t-1}, \beta_t I)$$

$$x_t = \sqrt{1-\beta_t} x_{t-1} + \sqrt{\beta_t} \cdot \epsilon, \quad \epsilon \sim \mathcal{N}(0, I)$$

$$\mathbb{D}x_t = (1-\beta_t) \mathbb{D}x_{t-1} + \beta_t \mathbb{D}\epsilon = 1$$

(assume that ϵ objects are normalized)

Now we introduce the reverse process:

$$p(x_0, x_1, \dots, x_T) = p(x_T) \prod_{t=1}^T p_\theta(x_{t-1}|x_t)$$

$\mathcal{N}(0, I)$

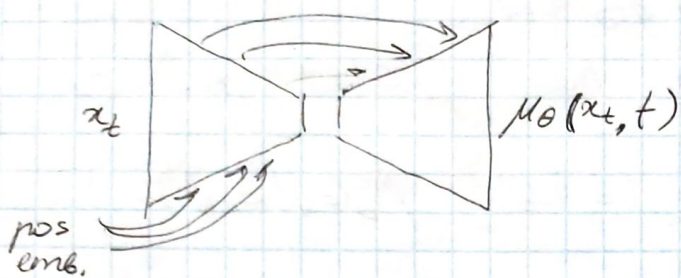
outputs from NN

(Σ_θ is output in general case)

$$p_\theta = \mathcal{N}(x_{t-1} | \mu_\theta(x_t, t), \Sigma_\theta(x_t, t))$$

$\mu_\theta(x_t, t)$ is transformation from image to image of the same size. In simplest case U-Net like architecture is used.

To consider time moment t we add positional embedding like in transformer.



Training:

$$\mathbb{E}_{x_0 \sim \text{Data}} \log p_\theta(x_0) \rightarrow \max_{\theta}$$

("noise schedule")
 β_t are chosen by user
 and aren't learned

So, since we learn only one denoising network, we learn it in a more stable way.

To efficiently learn we want ~~one~~ q to have the same structure comparing to p on the hidden variables.

$$q(x_1, \dots, x_T | x_0) = \prod_{t=1}^T q(x_t | x_{t-1}) = q(x_T | x_0) \prod_{t=2}^T q(x_t | x_t, x_0)$$

What are the expressions for $q(x_t | x_0)$ and $q(x_{t-1} | x_t, x_0)$?

Gaussian Linear Model

$$p(x) = \mathcal{N}(x | \mu, \Sigma)$$

$$p(y | x) = \mathcal{N}(y | Ax + b, \Gamma)$$

$$p(y, x) = p(y | x) \cdot p(x) \propto \exp \left(-\frac{1}{2} (x - \mu)^T \Sigma^{-1} (x - \mu) - \frac{1}{2} (y - Ax - b)^T \Gamma^{-1} (y - Ax - b) \right)$$

$$p(y) = \mathcal{N}(y | A\mu + b, \Gamma + A\Sigma A^T)$$

$$p(x | y) = \mathcal{N}(x | P(\Sigma^{-1}\mu + A^T \Gamma^{-1}(y - b)), P)$$

$$P = (\Sigma^{-1} + A^T \Gamma^{-1} A)^{-1}$$

$$q(x_t | x_0) = \mathcal{N}(x_t | \sqrt{\alpha_t} x_0, (1 - \alpha_t) I)$$

$$\bar{\alpha}_t = \prod_{s=1}^t \alpha_s, \quad \alpha_s = 1 - \beta_s$$

$$q(x_{t-1} | x_0, x_t) = \mathcal{N}(x_{t-1} | \tilde{\mu}_t(x_0, x_t), \tilde{\beta}_t I)$$

$$\tilde{\mu}_t(x_0, x_t) = \frac{\sqrt{\alpha_{t-1}} \beta_t x_0}{1 - \bar{\alpha}_t} + \frac{\sqrt{\alpha_t} (1 - \bar{\alpha}_{t-1})}{1 - \bar{\alpha}_t} x_t$$

$$\tilde{\beta}_t = \frac{1 - \bar{\alpha}_{t-1} \beta_t}{1 - \bar{\alpha}_t}$$

Training:

(consider single training object x_0)

$$\begin{aligned} \log p(x_0) &\geq \mathbb{E}_{q(x_1, \dots, x_T | x_0)} \log \frac{p(x_0, \dots, x_T)}{q(x_1, \dots, x_T | x_0)} = \\ &= \mathbb{E}_{q(x_1, \dots, x_T | x_0)} \log \left(\frac{p(x_T) \prod_{t=1}^T p_\theta(x_{t-1} | x_t)}{q(x_T | x_0) \prod_{t=2}^T q(x_{t-1} | x_t, x_0)} \right) = \\ &= \mathbb{E}_{q(x_1, \dots, x_T | x_0)} \left(\log \frac{p(x_T)}{q(x_T | x_0)} + \sum_{t=2}^T \log \frac{p_\theta(x_{t-1} | x_t)}{q(x_{t-1} | x_t, x_0)} + \log p(x_0 | x_1) \right) = \\ &= \underbrace{-\text{KL}(q(x_T | x_0) \| p(x_T))}_{\text{const}} + \sum_{t=2}^T \mathbb{E}_{q(x_t | x_0)} \text{KL}(q(x_{t-1} | x_t, x_0) \| p_\theta(x_{t-1} | x_t)) + \mathbb{E}_{q(x_1 | x_0)} \log p(x_0 | x_1) \rightarrow \max_{\theta} \end{aligned}$$

regularization terms

reconstruction term

$$\mathbb{E}_{q(x_t | x_0)} \text{KL}(q(x_{t-1} | x_t, x_0) \| p_\theta(x_{t-1} | x_t)) = - \text{can be computed as closed form expression since } q \text{ and } p \text{ are same normal distrib.}$$

$$\left| \begin{aligned} p_\theta(x_{t-1} | x_t) &= \mathcal{N}(x_{t-1} | \mu_\theta(x_t, t), \sigma_t^2 I) \\ q(x_{t-1} | x_t, x_0) &= \mathcal{N}(\tilde{\mu}_t(x_t, x_0), \tilde{\beta}_t I) \end{aligned} \right|$$

some predefined param (in simplest case)

$$= \mathbb{E}_{q(x_t | x_0)} \frac{1}{2\sigma_t^2} \|\tilde{\mu}_t(x_t, x_0) - \mu_\theta(x_t, t)\|^2 + \text{const} = \left| \begin{aligned} \text{rep. trick: } \epsilon &\sim \mathcal{N}(0, I) \\ x_t &= \sqrt{\alpha_t} x_0 + \sqrt{1 - \alpha_t} \epsilon \end{aligned} \right|$$

$$= \mathbb{E}_{x(\epsilon | 0, I)} \frac{1}{2\sigma_t^2} \|\tilde{\mu}_t(x_t(x_0, \epsilon), x_0) - \mu_\theta(x_t(x_0, \epsilon), t)\|^2 =$$

$$= \mathbb{E}_{x(\epsilon | 0, I)} \frac{1}{2\sigma_t^2} \left\| \frac{1}{\sqrt{\alpha_t}} (x_t(x_0, \epsilon) - \beta_t \epsilon) - \mu_\theta(x_t, t) \right\|^2 \quad \textcircled{=}$$

$\tilde{\mu}_t(x_t(x_0, \epsilon), x_0)$

Since on t near to T $\mu_\theta(x_t, t)$ should be almost noise image; on t near to 0 $\mu_\theta(x_t, t)$ should be almost predicted full-res image. And it's difficult goal for a single NN.

So, we use different parametrization:

$$\mu_\theta(x_t | t) = \frac{1}{\sqrt{\alpha_t}} \left(x_t - \frac{\beta_t}{\sqrt{1-\alpha_t}} \overset{\text{U-Net}}{\varepsilon_\theta(x_t, t)} \right)$$

- so, instead of producing a new image, we produce the noise, which we need to subtract from the current image. Plus we add normalization terms for good results on the final steps.

(there is also a choice to directly forecast noise to subtract from x_0).

$$\mathbb{E}_{N(\varepsilon|0, I)} \frac{1}{2\sigma_t} \cdot \frac{\beta_t^2}{\alpha_t(1-\alpha_t)} \|\varepsilon - \varepsilon_\theta(x_t(x_0, \varepsilon), t)\|^2 = L_{\text{ELBO}}$$

Since we have summation over thousands of elements (we have thousands of noising steps), we use such loss instead:

$$L_{\text{simple}} = \mathbb{E}_{N(\varepsilon|0, I), t \sim \text{Unif}(1, \dots, T)} \|\varepsilon - \varepsilon_\theta(x_t(x_0, \varepsilon), t)\|^2$$

Final algorithm:

Training $x_0 \sim \text{Data (batch)}$ ^{mini-}

$t \sim \text{Unif}(1, \dots, T)$

$\varepsilon \sim N(\varepsilon|0, I)$

make grad step for

$$\nabla_\theta \|\varepsilon - \varepsilon_\theta(\sqrt{\alpha_t} x_0 + \sqrt{1-\alpha_t} \varepsilon, t)\|^2$$

Inference: $x_T \sim \text{N}(x_T | 0, I)$

for t from T to 1 :

$$z \sim \text{N}(z | 0, I)$$

$$x_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left(x_t - \frac{\beta_t}{\sqrt{1-\alpha_t}} \epsilon_\theta(x_t, t) \right) + \sqrt{1-\alpha_t} z$$

x_0 - the result

We then can do some tricks/improvements like optimizing $\mathcal{L}_{\text{simple}} + \lambda \mathcal{L}_{\text{ELBO}}$, choose noise schedule, how to sample time moments, etc.

Adversarial attacks

Attack:

$$x, \hat{x} = x + z, \quad z \text{ is small}$$

(almost not noticeable perceptually)
 $y(x) \neq y(\hat{x})$ - by slightly changing x we fool the NN y .

And z may be chosen for arbitrary class you want.
We need to defend from attacks.

$$\text{Targeted attack: } \begin{cases} p(x, x+z) \rightarrow \min_z \\ y(x+z) = \text{target} \end{cases}$$

$$\text{Non-targeted attack: } \begin{cases} p(x, x+z) \rightarrow \min_z \\ y(x+z) \neq y(x) \end{cases}$$

L-BFGS

targeted: $\downarrow$ ^{some small const.} $\|z\|^2 \rightarrow \min$ while $y(x+z) = \text{target}$

$$L = \|z\|^2 - c \log p(y = \text{target} | f(x+z)) \rightarrow \min_z$$

compute $\nabla_z L$ and optimize using some opt. algorithm.

non-targeted:

$$\|z\|^2 + c \log p(y = y(f(x)) | f(x+z)) \rightarrow \min_z$$

Fast gradient sign method (FGSM)

$$\begin{cases} \tilde{f}(x+z) = \log p(y = y(f(x)) | f(x+z)) \rightarrow \min_z \\ \|z\| \leq \epsilon \end{cases} \quad \leftarrow \text{non-targeted}$$

$$\tilde{f}(x+z) \approx \tilde{f}(x) + \nabla \tilde{f}(x)^T z$$

$$\begin{cases} \tilde{f}(x) + \nabla \tilde{f}(x)^T z \rightarrow \min_z \\ \|z\|_\infty \leq \epsilon \end{cases} \Leftrightarrow \begin{cases} \nabla \tilde{f}(x)^T z \rightarrow \min_z \\ |z_i| \leq \epsilon \quad \forall i \end{cases}$$

$$\max(|z_1|, \dots, |z_d|)$$

$$\text{KKT: } \begin{cases} z \in \text{Argmin}_z (\nabla f(x)^T z + \sum_i \lambda_i (|z_i| - \epsilon)) \\ |z_i| \leq \epsilon \\ \lambda_i \geq 0 \\ \lambda_i (|z_i| - \epsilon) = 0 \end{cases}$$

$$\lambda_i = 0 \quad (\nabla f(x))_i: z_i \rightarrow \min_z, \quad z_i \rightarrow -\infty$$

$$\lambda_i \neq 0 \Rightarrow |z_i| = \epsilon$$

$$\begin{matrix} z_i = \epsilon \\ z_i = -\epsilon \end{matrix} \quad \begin{matrix} (\nabla f(x))_i \cdot \epsilon \\ -(\nabla f(x))_i \cdot \epsilon \end{matrix} \Rightarrow \boxed{z_i = -\epsilon \cdot \text{sign}((\nabla f(x))_i)}$$

↙ in case of non-targeted attack

$$\hat{x} = x + \epsilon \cdot \text{sign}(\nabla f(x))$$

if class label didn't change, just continue this operation iteratively.

Deep Fool

$y \in \{-1, 1\}$, $f(x)$ - score for $y(x)$ ($y(x) = \text{sign}(f(x))$)

$$f(x+z) \approx f(x) + \nabla f(x)^T z$$

$$\begin{cases} \|z\|_2^2 \rightarrow \min_z \\ f(x) + \nabla f(x)^T z = 0 \end{cases}$$

$$L(z, \mu) = \|z\|_2^2 + \mu (f(x) + \nabla f(x)^T z)$$

$$\text{KKT: } \begin{cases} \nabla_z L(z, \mu) = 0 = 2z + \mu \nabla f(x) \\ f(x) + \nabla f(x)^T z = 0 \end{cases}$$

$$z = -\frac{\mu \nabla f(x)}{2}$$

$$f(x) - \frac{\mu}{2} \|\nabla f(x)\|_2^2 = 0 \Rightarrow \mu = \frac{2f(x)}{\|\nabla f(x)\|_2^2}$$

$$\boxed{z = -\frac{f(x)}{\|\nabla f(x)\|_2^2} \cdot \nabla f(x)} \quad \text{and can do iteratively}$$

$y \in \{1, 2, \dots, K\}$ scores for other classes

$$f_{y(x)}(x) \geq f_k(x) \quad \forall k \neq y(x)$$

$$\begin{cases} \|z\|^2 \rightarrow \min_z \\ \exists k \ f_{y(x)}(x+z) = f_k(x+z) \rightarrow \exists k \ f_{y(x)}(x) + \nabla f_{y(x)}(x)^T \cdot z = \\ \qquad \qquad \qquad = f_k(x) + \nabla f_k(x)^T \cdot z \\ \exists k \ f_{y(x)}(x) - f_k(x) + (\nabla f_{y(x)}(x) - \nabla f_k(x))^T \cdot z = 0 \end{cases}$$

~~let~~ $z = \min_k z_k,$

$$z_k = - \frac{(f_{y(x)} - f_k(x))}{\|\nabla f_{y(x)}(x) - \nabla f_k(x)\|^2} \cdot (\nabla f_{y(x)}(x) - \nabla f_k(x))$$

and do it iteratively unless class label is changed

Carlini-Wagner

$$L = \|z\|^2 + c \mathcal{F}(x, z) \rightarrow \min_z$$

targeted

$$\mathcal{F}(x, z) = \max \left(\max_{y \neq \text{target}} \log p(y | f(x+z)) - \log p(y = \text{target} | f(x+z)), -\epsilon \right)$$

non-targeted

$$\mathcal{F}(x, z) = \max \left(\log p(y = y(x) | f(x+z)) - \max_{y \neq y(x)} \log p(y | f(x+z)), -\epsilon \right)$$

compute $\nabla_z L$ and optimize using some optimizer.

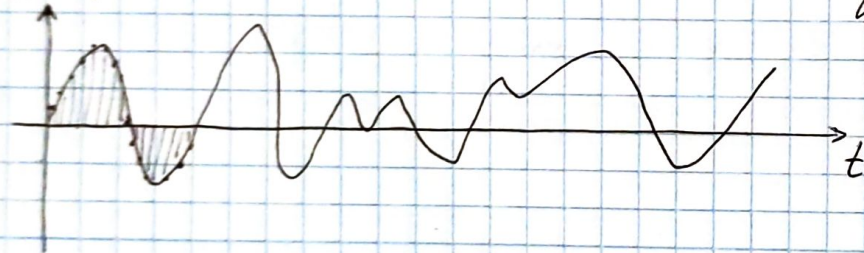
Audio processing. Text-To-Speech

Applications:

- ~~audio~~ speech recognition
- voice ~~speech~~ synthesis
- audio ~~sequence~~ classification
- audio segmentation
- audio denoising
- etc.

How to input audio into NN?

wav:



Usual sample rate: 44kHz

(for processing usually reduced to like 16kHz)

Before putting into NN, we first do a preprocessing of the raw audio signal into more representable form.

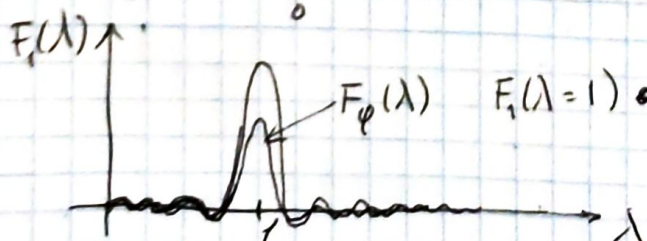
We have basic properties of a sine wave:

Frequency ("pitch"), amplitude ("loudness") and phase. ~~the phase is not needed~~

We want to recover these properties from an arbitrary parts of audio signals. The phase usually isn't needed, but in some tasks it is, and it's hard to recover from a histogram.

theoretically should be from $-\infty$ to $+\infty$

$$F_1(\lambda) = \int_0^{\infty} \sin(x) \sin(\lambda x) dx$$



we want to get rid of this during frequency/amplitude recovering

$$F_{\varphi}(\lambda) = \int \sin(x+\varphi) \sin(\lambda x) dx$$

In Fourier transform:

$$F_1(\lambda) = \int \sin(x+\varphi) \sin(\lambda x) dx$$

$$F_2(\lambda) = \int \sin(x+\varphi) \cos(\lambda x) dx$$

$$F_1^2(\lambda) + F_2^2(\lambda) \quad - \text{doesn't depend on } \varphi$$

To recover φ itself: $\varphi = \text{atan}\left(\frac{F_1(\lambda)}{F_2(\lambda)}\right)$

Due to computational reasons we compute in $\mathbb{C}$:

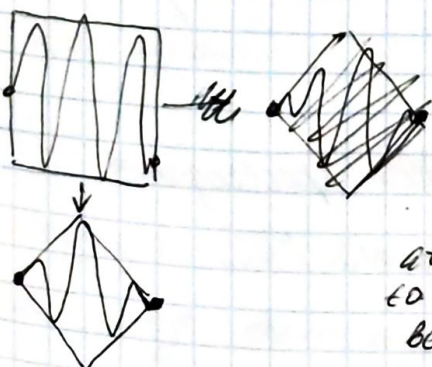
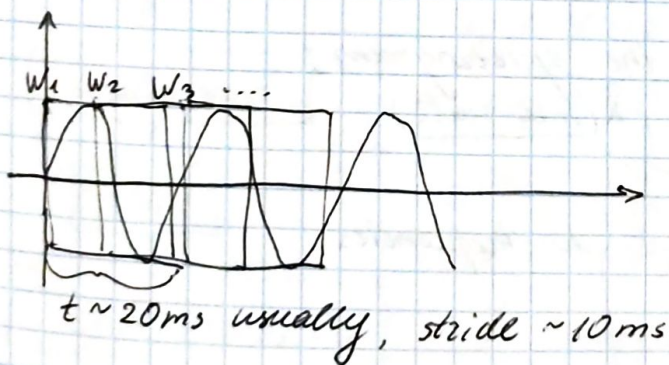
$$F(\lambda) = \int \sin(x+\varphi) \exp(i\lambda x) dx$$

So, we choose set of λ , ~~and~~ and compute $F(\lambda_i)$ over ~~the~~ specified time intervals.

$$F(\lambda_i) = \int_{t_0}^{t_1} f(x) \exp(i\lambda x) dx$$

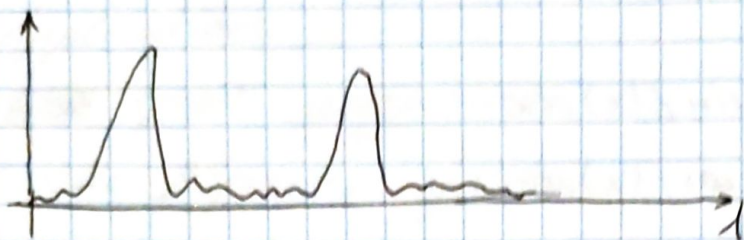
Then amplitude = $|F(\lambda_i)|^2$

Practically we split the input ~~sequence~~ audio into overlapping windows, for which $F(\lambda_i)$ is computed

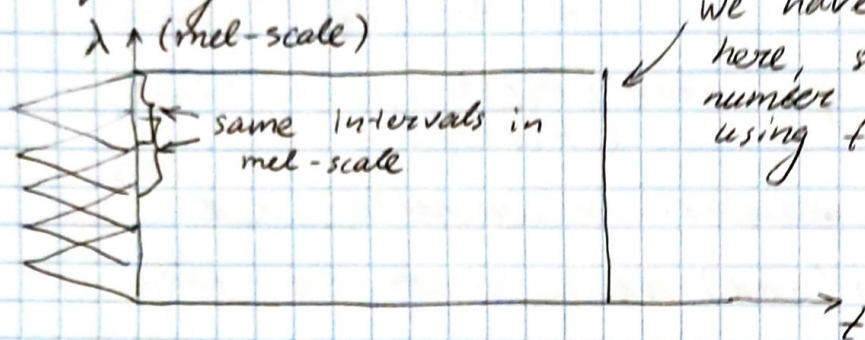


- multiply window by triangular function
because FT is applied to periodic functions, so the function should ~~be~~ have the same values at the start and at the end in order to not have abnormal frequencies because of the gap.

$F(\lambda)$

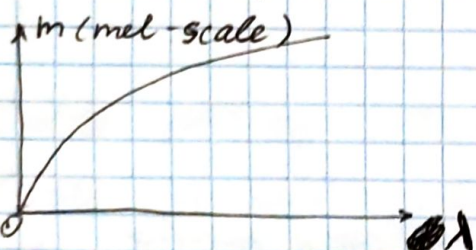


Spectrogram:



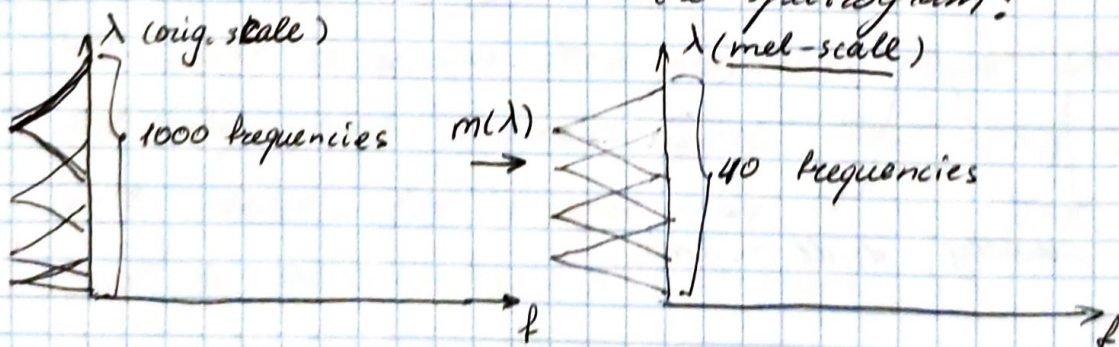
we have too much frequencies here, so we sum up into less number of frequencies (say, 40) using triangular window.

But the pitch we hear is actually a logarithm of the corresponding frequency. So, we work in mel-scale, which is almost logarithmic.



$$m \approx 2595 \cdot \log_{10}\left(1 + \frac{\lambda}{700}\right) \approx \frac{\lambda}{700} \text{ for } \frac{\lambda}{700} \text{ near to } 0.$$

We have such transform of the spectrogram:



log-mel-spectrogram:

We actually use $\log(|F(\lambda)|)$ in the spectrogram, because of how human hears loudness.

MFCC (mel-frequency cepstrum coefficient)

For a given time ~~interval~~^{frame} it's a discrete fourier transform of log-mel-spectrum:

~~MFCC is the log of the magnitude of the DFT of the log-mel-spectrum~~

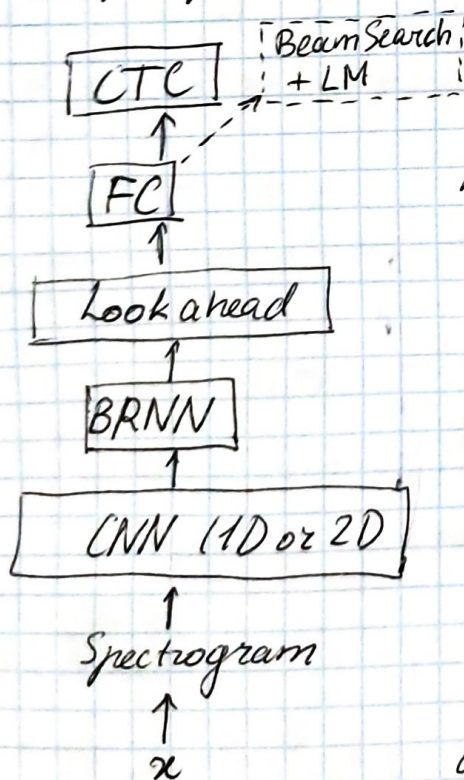
$$MFCC(s) = |DFT(\log(|FT(f, m(\lambda))|), s)|$$

Helpful for analyzing signals with periodic frequencies.

In practice we may concatenate all these features (original spectrum, log-mel-spectrum, MFCC) and use it as inputs to a NN.

Also, we may consider the spectrogram (log-mel-spectrogram) as an image and use CNN on it.

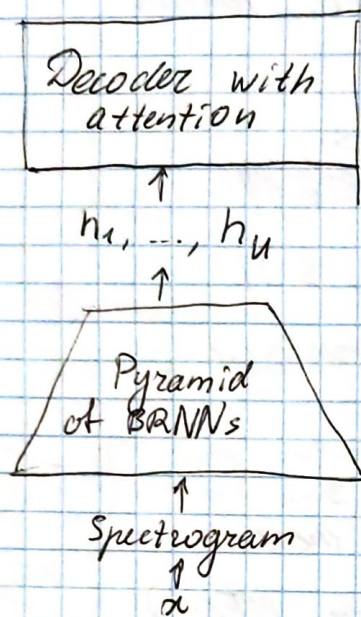
Deep Speech 2



Takes audio as input, then produces spectrogram. When applying CNN, we ~~do~~ use convolutions, which captures big part of the spectrum for each time frame. Lookahead layer is used in case of unidirectional RNN to have future context. After FC layer we have pdf over language symbols, which ~~is~~ is used

to compute CTC loss on training and to predict words using beam search with LM (lang. model) on testing.

LAS (Listen, Attend and Spell)



Text - To-Speech (TTS)

ASR: Raw audio $\rightarrow$ Spectrogram $\xrightarrow{NN}$ Text
TTS: Text $\xrightarrow{NN}$ Spectrogram $\xrightarrow{\text{Vocoder}}$ Raw audio

Griffin-Lim vocoder

A classic algorithm from 80. Doesn't have any parameters.

Input: $P = |STFT(x)|$ - short time Fourier transform of raw audio x (P is spectrum $P \in \mathbb{R}^K$)

Output: $\tilde{x} \approx x$

The problem here is that we don't know phases φ_k and we need to somehow recover them.

$\varphi \sim \mathcal{R}[0, 2\pi]$ - random samples from uniform distr. on $[0, 2\pi]$, K - number of freq.

$$\tilde{x} = iSTFT(P \cdot \exp(i\varphi))$$

function time $\leftarrow$ freq. $1/2\pi$ (angular freq.) $\leftarrow$ window func.

$$F(\tau, \omega) = STFT(f, \tau, \omega) = FT(f \cdot w_\tau, \omega) = |W_\tau(x)| = |W(x - \tau)|$$

(complex) - function of τ and ω

$$f(t) = iSTFT(F, t) = \frac{1}{W(t)}$$

func. prod.

$$F(\tau, \omega) = STFT(f|\tau, \omega) = FT(f \cdot w_\tau | \omega) = |W_\tau(t) = W(t - \tau)| = \int_{-\infty}^{\infty} f(t) W(t - \tau) \cdot e^{-i\omega t} dt$$

- complex function of τ and ω

$$f(t, \tau) = iSTFT(F|t, \tau) = iFT(F \cdot \frac{1}{W(t-\tau)} | t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} \frac{F(\tau, \omega)}{W(t-\tau)} \cdot e^{i\omega t} d\omega = \frac{1}{2\pi \cdot W(t-\tau)} \int_{-\infty}^{\infty} F(\tau, \omega) \cdot e^{i\omega t} d\omega$$

$\bullet$ - real-valued function defined where $W(t-\tau) \neq 0$.

$$\int_{-\infty}^{\infty} W(x) dx = 1 \quad \text{- requirement}$$

$$f(t) \cdot w(t-T) = \text{IFT}(F|t)$$

Note, that F is complex valued, containing amplitude in $|F|$ and phase $\arg F$. On practice we ^{usually} only have $|F|$. So, we may choose phase $\varphi(\tau, \omega)$ and do ~~the inverse Fourier transform~~ $\text{IFT}(|F|e^{i\varphi}|t)$.

On reconstruction we do iFT for each t at needed sample rate, taking nearest τ from our discrete set.

Griffin-Lim vocoder

Input: $|F| \in \mathbb{R}^{K \times T_F}$ - histogram of some input ^{audio} ~~signal~~ f , where K is number of frequencies (say, 1000) and T_F number of frames (say, 100 per second $\times n$ seconds)

Output: $\tilde{f} \in \mathbb{R}^T$, where $\tilde{f} \approx f$ and $|F| = |\text{STFT}(f)|$
 - $\tilde{f}$ - reconstructed audio with T samples (say, 44100 ~~per~~ per second $\times n$ seconds)

The problem here is that we don't have phase information in $|F|$.

So, we first sample them from uniform distr.

$$\varphi \sim \text{Unit}^{K \times T} [0, 2\pi) \quad \varphi \in \mathbb{R}^{K \times T}$$

Then do iteratively until convergence:

for ϵ step j :

$$\tilde{F} = \text{ISTFT}(|F| \cdot e^{i\varphi})$$

$$\varphi = \arg \text{STFT}(\tilde{F})$$

Note, that $T_F \ll T$, so we ^{must} do interpolation between φ from nearest windows, when performing ISTFT.

Quality of this algorithm is bad.

WaveNet vocoder

Now, let's use x instead of f for raw audio.

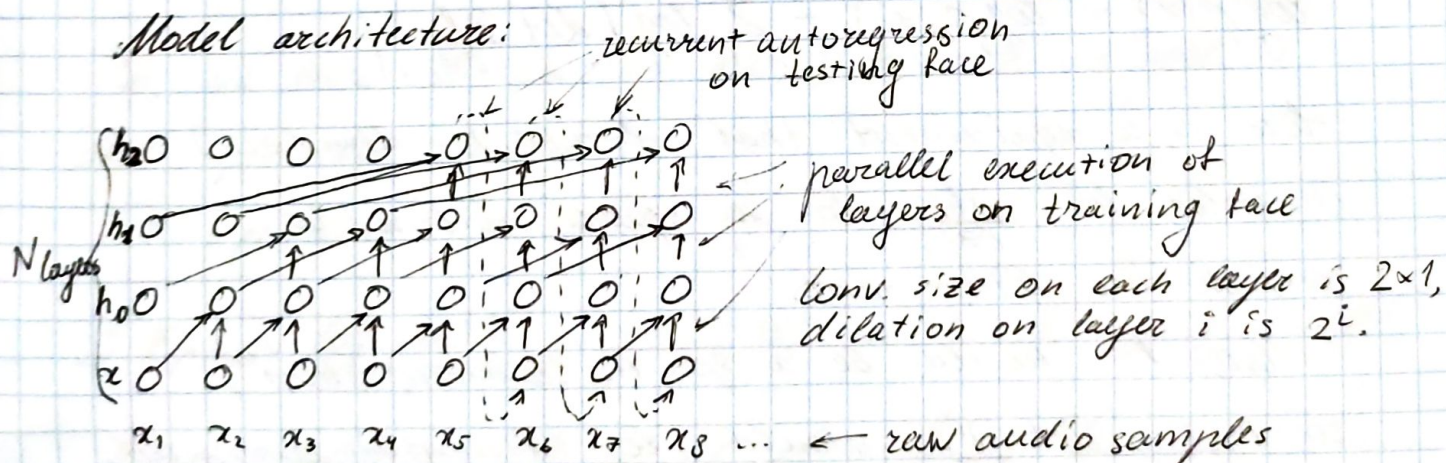
$\{x_1, \dots, x_T\} = x$ - samples.

$$p(x_1, \dots, x_T) = \prod_{t=1}^T p(x_t | x_{<t}, \theta) \quad \text{- autoregressive model.}$$

Here the problem is that computations are sequential and we need to do them for each sample, which is too long.

~~increasing~~ But we may parallelize on training face,

Model architecture:



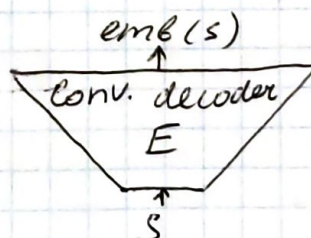
$$h_i = \sigma(W_{gi} * h_{i-1} + V_{gi} * \text{emb}(s)) \odot \tanh(W_{fi} * h_{i-1} + V_{fi} * \text{emb}(s)),$$

Where s is spectrogram (with low sample rate) and

$\text{emb}(s)$ is matrix of embeddings

for each time moment in the

output sequence.



So, we have $W_{gi}, V_{gi}, W_{fi}, W_{gi}$ and ~~conv. decoder~~ weights of conv. decoder E as learnable parameters. ~~Here~~ On the output at each time moment we may have just samples (some real numbers in range $[0, 1]$ or $[-1, 1]$) and use squared error as a loss, or we may have softmax distribution over possible sample values (for ex., we will have only 256 possible values for 8-bit encoding) and use cross entropy.

Parallel WaveNet vocoder

Normalizing flows:

some prior distr., for ex. $N(0, I)$

$$z_0 \sim p_0(z_0), \quad z_1 = f_1(z_0, \theta_1), \quad \dots \quad z_N = f_N(z_{N-1}, \theta_N) = x$$

Must be invertible

$$\rightarrow z_0 = f_1^{-1}(z_1, \theta_1) \quad \dots \quad z_{N-1} = f_N^{-1}(z_N, \theta_N)$$

This way we have access to our ~~prior~~ data distribution.

$$\log p(z_i) = \log p(z_{i-1}) - \log \left| \det \frac{\partial f_i}{\partial z_{i-1}} \right| \quad \leftarrow \text{Jacobian}$$

$$\log p(x) = \log p_0(z_0) - \sum_{i=1}^N \log \left| \det \frac{\partial f_i}{\partial z_{i-1}} \right|$$

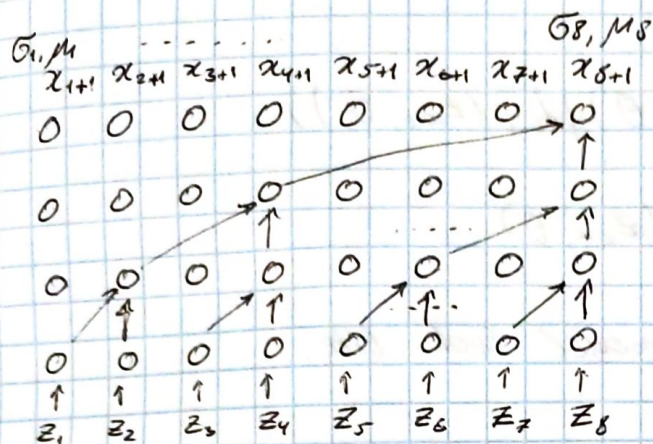
Here is a requirement that $\dim(z_i) = \dim(z_{i-1})$ in order for $\frac{\partial f_i}{\partial z_{i-1}}$ to be square matrix to compute the determinant.

Also, f_i should be a specific transformation, so that $\det \frac{\partial f_i}{\partial z_{i-1}}$ would be efficiently computed.

Now we can directly optimize the likelihood:

$$\sum_{x \in \text{training}} \log p(x | \theta) \rightarrow \max$$

Parallel WaveNet:



it also outputs σ_i and μ_i for computing x_{i+1} . And seems this convolutional architecture is only used for producing σ_i and μ_i .

$$z_t \sim p(z) \text{ (usually } p = \mathcal{N}(0, I)\text{)}$$

it's standard deviation,

$$x_t = z_t \cdot \sigma_{t-1}(z_{<t}, \theta) + \mu_{t-1}(z_{<t}, \theta)$$

$$\log p(x|z) = \log p(z) - \log \left| \det \frac{\partial x}{\partial z} \right|$$

lower-triangular matrix, cause x_t depends only on $z_{<t}$

$$\frac{\partial x_t}{\partial z_t} = \sigma_{t-1}(z_{<t}, \theta) \Rightarrow \log p(x, z) = \log p(z) - \sum_{t=1}^T \log \sigma(z_{<t}, \theta)$$

$$z_1 = x_1$$

$$z_2 = \frac{x_2 - \mu_1(z_1, \theta)}{\sigma_1(z_1, \theta)}$$

...

$$z_t = \frac{x_t - \mu_{t-1}(z_{<t}, \theta)}{\sigma_{t-1}(z_{<t}, \theta)}$$

inversion process

- not parallel $\Rightarrow$ slow computation of $p(z)$ (cause we need to do the inverse transformation to obtain z) and slow backprop.

So, we take distribution $p_T(x)$ from WaveNet and optimize KL div.:

$$KL(p(x|z) \parallel p_T(x)) \rightarrow \min_{\theta} \quad \textcircled{1}$$

$$\textcircled{2} \mathbb{E}_{p(x|z)} \log \frac{p(x|z)}{p_T(x)} = \mathbb{E}_{p(x|z)} \log p(x|z) - \mathbb{E}_{p(x|z)} \log p_T(x)$$

$$\mathbb{E}_{p(x|z)} \log p(x|z) = \mathbb{E}_{p(z)} \left(\log p(z) - \sum_{t=1}^T \log \sigma(z_{<t}, \theta) \right) =$$

$$= \underbrace{\mathbb{E}_{p(z)} \log p(z)}_{= \text{const}} - \sum_{t=1}^T \mathbb{E}_{p(z)} \log \sigma(z_{<t}, \theta)$$

so, now we don't need to compute it

estimated using Monte-Carlo

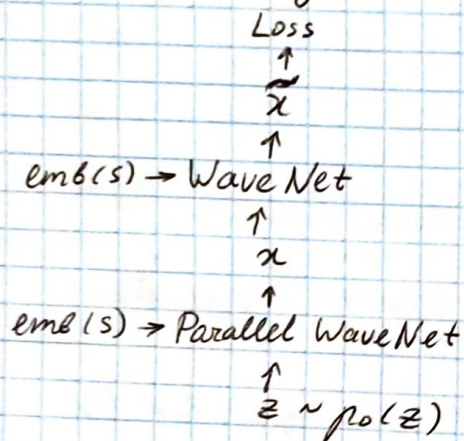
$$\mathbb{E}_{p(x|z)} \log p_T(x) = \sum_{t=1}^T \mathbb{E}_{p(x|z)} \log p_T(x_t | x_{<t}) =$$

$$= \sum_{t=1}^T \mathbb{E}_{p(x_{t-1}|z)} \cdot \mathbb{E}_{p(x_t|x_{t-1}, z)} \log p_T(x_t | x_{t-1})$$

$$= \sum_{t=1}^T \mathbb{E}_{p(z_{t:t})} \log p_T(h_t(z_{t:t}, \theta) | h_{t-1}(z_{t:t}, \theta))$$

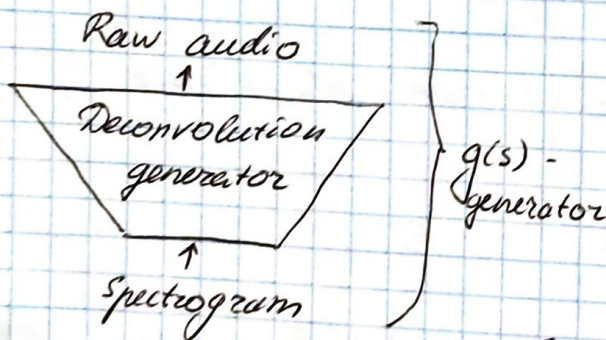
$$h_t(z_{t:t}, \theta) = z_t \sigma(z_{t:t}, \theta) + \mu(z_{t:t}, \theta)$$

So, finally on training the model looks like:



HiFi-GAN

Here we are conditioned on mel-spectrogram.
And we don't use any noise as an input.



Then we have some ^{learnable} function $d(x)$ as discriminator.

And the optimization is following.

$$\mathbb{E}_{s \sim \text{Data}} (d(g(s)) - 1)^2 \rightarrow \min_g$$

So it's not a generative model.

$$\mathbb{E}_{x \sim \text{Data}} (d(x) - 1)^2 + \mathbb{E}_{s \sim \text{Data}} d(g(s))^2 \rightarrow \min_d$$

Reinforcement learning

Problem statement

ML/DL:

$\{x_i, y_i\}_{i=1}^N$ $f(x, \theta)$ - predictor

$$F(\theta) = \frac{1}{N} \sum_{i=1}^N L(y_i, f(x_i, \theta)) \rightarrow \min_{\theta}$$

$f(x, \theta)$ is differentiable

RL:

Reward function ^{usually} isn't differentiable.

There is no right answer.

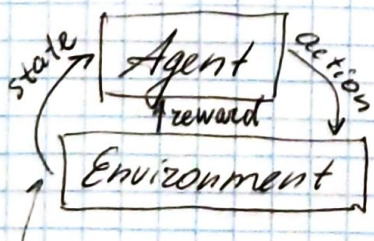
No training dataset. It's impossible to collect a dataset, representing all the states of a game.

Need to somehow estimate the quality of a particular position in the game.

And the same for other applications, like in robot control systems.

You need to simultaneously learn the agent and collect the dataset (explore the environment).

If you have data and a loss function for it, it's better to use classic ML/DL to solve the problem, cause it's just more effective.



Environment is described by state $s \in S$ (like state of a chessboard).

$a \in A$ - ~~agent~~ action (performed at each time moment)

but there are cases when the agent can't observe the full environment, but only a part of it (for now we discuss only fully observable case)

$\pi(a|s)$ - policy (may be deterministic or stochastic, i.e. may be either a value for a given s or a pdf)

property of the environment (so, we say what action to perform, then the environment "decides" in what state to go according to prob. distr.)
 $p(s'|s, a)$ - transition probability - move the environment to the next state given current state and an action.
 (also can be deterministic or stochastic).

$r(s, a) \in \mathbb{R}$ - reward - scalar function - want to maximize by optimizing $\pi(a|s)$ (may also be stochastic)

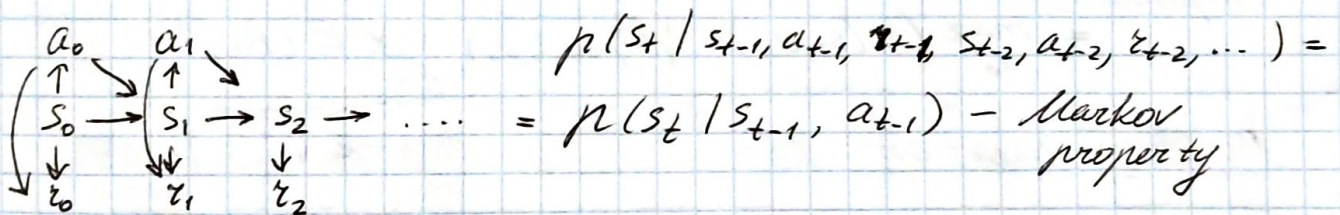
environment initialization (may be chosen from a p.d.f over S)
 $\tau = \{s_0, a_0, r_0, s_1, a_1, r_1, \dots\}$ - trajectory

We want to learn $\pi(a|s)$ by maximizing accumulative reward:

$$R_t = r_t + r_{t+1} + r_{t+2} + \dots$$

Goal: $E[R_0] \rightarrow \max_{\pi}$ expectation w.r.t. all stochasticity we have in the environment (w.r.t. $\pi(a|s)$, $p(s'|s, a)$ for all time moments, $r(s, a)$ (if it's stochastic))

Markov Decision Process:



In fact we have the following reward (cause we can't sum to infinity):

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + \dots, \quad \gamma \in (0, 1) - \text{discount}$$

Since $\gamma < 1$ this sum will converge.

Also it makes taking the reward at current time moment to be better than in next time moment.

(it will be preferable to go directly to desired state as fast as we can)

We may choose just some sequence $\{\gamma_t\}$ whose sum is converging to a finite value.

Value Iteration

Bellman function (value function):

$V^*(s) := \mathbb{E}(R_t | s_t = s)$ - expectation of cumulative reward if we start from time moment t , but t doesn't

matter, cause the reward sum is infinite.

$$V^*(s) = \max_{\pi} V^{\pi}(s)$$

$$R_t = r_t + \gamma r_{t+1} + \dots = r_t + \gamma R_{t+1}$$

Bellman equation

$$\begin{aligned} V^*(s_0) &= \max_{\pi} V^{\pi}(s_0) = \left| \begin{array}{l} \pi - \text{deterministic} \\ \text{policy} \end{array} \right| = \\ &= \max_{\substack{a_0, a_1, a_2, \dots \\ \uparrow \uparrow \uparrow \\ \text{independent}}} \mathbb{E}(R_0 | s_0 = s_0) = \max_{a_0, a_1, \dots} (r(s_0, a_0) + \gamma \mathbb{E}_{p(s_1 | s_0, a_0)} \mathbb{E}(R_1 | s_1 = s_1)) = \\ &= \max_{a_0} (r(s_0, a_0) + \gamma \mathbb{E}_{p(s_1 | s_0, a_0)} \underbrace{\max_{a_1, a_2, \dots} \mathbb{E}(R_1 | s_1 = s_1)}_{= V^*(s_1)}) \end{aligned}$$

Bellman equation:

$$V^*(s) = \max_a (r(s, a) + \gamma \mathbb{E}_{p(s' | s, a)} V^*(s')) \quad \forall s$$

~~$V^*(\pi(a|s))$~~ in case of deterministic π

$$a_0^* = \operatorname{argmax}_a (r(s_0, a_0) + \gamma \mathbb{E}_{p(s_1 | s_0, a_0)} V^*(s_1)) =$$

$= \pi^*(a_0 | s_0)$ - depends only on state.

So for the same states at different time moments we would get the same optimal action.

And for stochastic $\pi(a|s)$ the Bellman equation will remain the same. But $\pi^*(a_0 | s_0)$ will be deterministic policy.

Value Iteration algorithm:

We consider the case when S is a ^{finite} discrete set.

$$V_{\text{new}}(s) = \max_a (r(s, a) + \gamma \mathbb{E}_{p(s'|s, a)} V_{\text{old}}(s'))$$

1. Somehow initialize $V(s)$ for all possible s .
2. ~~Until~~ Until convergence do:

for s in S :

$$V(s) = V_{\text{new}}(s)$$

3. $\pi^*(a|s) = \arg\max_a (r(s, a) + \gamma \mathbb{E}_{p(s'|s, a)} V(s'))$

Example:

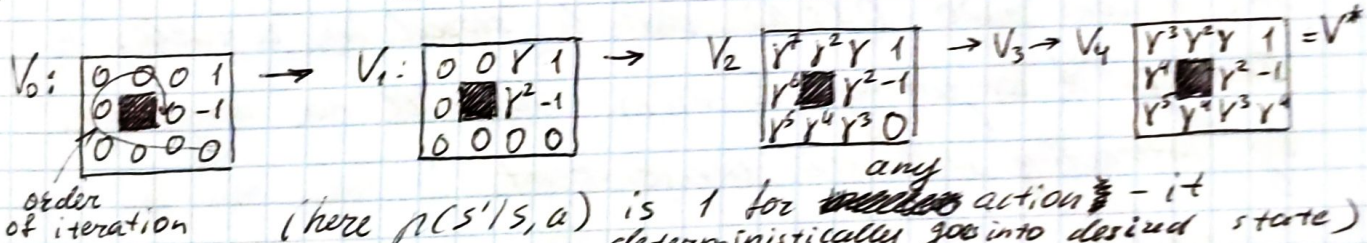
rewards

3	0	0	1
0	0	0	-1
0	0	0	0

prohibited cell

terminal states

At each time moment we can go in one of the 4 directions. Rewards are independent from actions.



The optimal policy is to go to the cell with the maximum reward.

Problems of the algorithm: $\pi^*(a|s)$ requires additional computation at each time moment (need to ~~find~~ ^{compute} $\mathbb{E}_{p(s'|s, a)} V(s')$)

In case of huge (or continuous) space S

we can't compute $\mathbb{E}_{p(s'|s, a)} V(s')$.

We're not able usually even ~~more~~ to restore $p(s'|s, a)$ as a function.

Also in practical cases usually we can't store values for all possible states.

But this algorithm converges to the global maximum.

Q-learning

quality function

$$Q^*(s, a) := \mathbb{E}(R_t | s_t = s, a_t = a)$$

$$Q^*(s, a) = \max_{\pi} Q^{\pi}(s, a)$$

$$Q^*(s, a) = r(s, a) + \gamma \mathbb{E}_{\pi(s'|s, a)} \max_{a'} Q^*(s', a') \quad \forall s, a$$

Bellman equation for Q function $\uparrow$

$$\pi^*(a|s) = \operatorname{argmax}_a Q^*(s, a) \quad \forall s$$

$$V^{\pi}(s) = \mathbb{E}_{\pi(a|s)} Q^{\pi}(s, a)$$

$$V^*(s) = \max_a Q^*(s, a)$$

How to calculate $\mathbb{E}_{\pi(s'|s, a)} \max_{a'} Q^*(s', a')$

We are still dealing with ^{small} finite number of states and actions. So $Q(s, a)$ can be stored in a table.

Instead of computing $\mathbb{E}_{\pi(s'|s, a)}$, we will do sampling by interacting with the environment.

We need to find $Q(s, a)$, that satisfies the Bellman ~~EQN~~ equation for $\forall s, a$. So, we solve a set of non-linear equations. We may reformulate it as optimization:

$$F(Q) = \frac{1}{|S| \cdot |A|} \sum_{s, a} \underbrace{(Q(s, a) - r(s, a) - \gamma \mathbb{E}_{\pi(s'|s, a)} \max_{a'} Q(s', a'))^2}_{\text{huge sum}} \rightarrow \min_Q$$

Let's minimize by kind of ~~SGD~~ SGD: we will sample batch of $\{s_i, a_i\}_{i=1}^B$ and minimize $F(Q)$ for this batch.

$$\sim (s, a, r, s')$$

$$Q_{\text{new}}(s, a) = Q_{\text{old}}(s, a) - \alpha$$

We sample $\sim (s, a, \text{reward})$ then $s' \sim p(s' | s, a)$

$$Q_{\text{new}}(s, a) = Q_{\text{old}}(s, a) - \alpha (Q_{\text{old}}(s, a) - r(s, a) - \gamma \mathbb{E}_{p(s'|s, a)} \max_{a'} Q(s', a'))$$

Note, that $\mathbb{E}_{p(s'|s, a)} \max_{a'} Q(s', a')$ is a stochastic estimate by several samples (usually just one)

But here we have $\max_{a'} Q_{\text{old}}(s', a')$, which leads to overestimation

$\max_{a'} Q_{\text{old}}(s', a')$ will output ~~the current value~~ not $\max Q^*$, but the value for current estimation of Q .

$$Q_{\text{new}}(s, a) = (1 - \alpha) + \alpha y(s, a)$$

$y(s, a)$ is sampled time to time (like ~~per~~ per 100 steps) -

- it helps to overcome the problem with overestimation.

Exploration vs exploitation dilemma:

If we go according to some ~~current~~ current $Q(s, a)$, we won't explore more environment and search global optimum. (we don't try ~~a~~ locally unoptimal actions according to $Q(s, a)$).

So we need to artificially add exploration property to our algorithm.

ϵ -greedy:

$$\pi(a|s) \sim \begin{cases} \arg\max_a Q(s, a) & \text{with prob } 1 - \epsilon \\ \text{random action} & \text{with prob } \epsilon \end{cases}$$

So, at the first iterations we start from high ϵ (say, $\epsilon = 1$) and ~~all~~ gradually reduce it to 0 with iterations.

Softmax:

$$\pi(a|s) \sim \text{Softmax}_a \left(\frac{Q(s, a)}{\tau} \right)$$

With $\tau \rightarrow 0$ $\pi(a|s) \rightarrow \arg\max_a (Q(s, a))$

$\tau \rightarrow \infty$ $\pi(a|s) \sim \text{Unif}[A]$

With softmax we better use our $Q(s,a)$ function.

Finally, ^{one step of} Q -learning is as following:

1. sample some state s
2. sample an action ~~use~~ for this state using ϵ -greedy or Softmax exploration strategy
3. sample s' according to $p(s'|s,a)$. Note, that it's enough to make only one sample as a stochastic estimate.

Also: ~~using~~ sampling s' is generally an expensive operation, involving some simulations in our environment. It's why we use $Q(s,a)$ instead of $V(s)$, because for $Q(s,a)$ we may do just one sample at each step, while in order to update $V(s)$ we need to do a sample for each possible action ~~due~~ in order to find $\arg\max_a$.

4. update $Q(s,a)$

(5.) update $q(s,a)$ each n steps.

DQN (Deep Q-learning) (Deep Q Network)

Just Q -learning requires storing a table for each ~~state~~ state s in S , which is not possible for huge or infinite S .

So, let's just make Q to be a parametrized function:

$$Q(s,a|\theta) : S \times A \rightarrow \mathbb{R} \quad - \text{some NN}_{\theta}$$

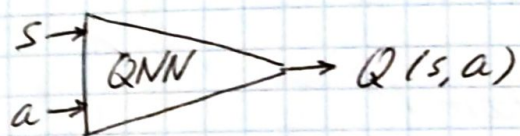
$$F(\theta) = \frac{1}{|S| \cdot |A|} \sum_{s,a} (Q(s,a|\theta) - r(s,a) - \gamma \mathbb{E}_{p(s'|s,a)} \max_{a'} Q(s',a'|\theta))^2$$

$\rightarrow \min_{\theta}$

Again:

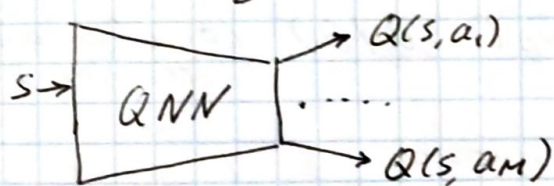
1. Sample s
2. Sample a , depending on s
3. Get $r(s, a)$ and s' from $p(s'|s, a)$
4. $Q_{\text{new}} = Q_{\text{old}} - \alpha (Q(s, a | Q) - y(s, a)) \cdot \nabla_{\theta} Q(s, a | \theta)$
- we may use any other optimization algorithm

Naive choice for $Q(s, a | \theta)$:



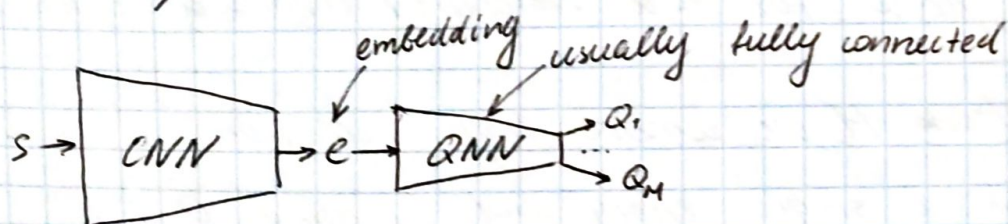
During computation of $\max_{a'} Q(s, a' | \theta)$, we will need to do full forward propagation for $\forall a \in A$. - inefficient.

So, usually:



This approach require just one forward propagation.

If s is described by an image, we usually use some pre-trained CNN:



DQN scheme:

Initialize memory M (replay buffer)

Initialize QNN θ

for episodes from 1 to P :

initialize s_0

for $t = 0, \dots, T$

$a_t \sim \epsilon$ -greedy or Softmax

obtain r_t and $s_{t+1} = s' \sim p(s' | s_t, a_t)$

$(s_t, a_t, r_t, s_{t+1}) \rightarrow M$

$(s, a, r, s') \sim M$ - sample from M

$$y = \begin{cases} r, & \text{if } s' \text{ is } \text{terminated state} \\ r + \gamma \max_{a'} Q(s', a' | \hat{\theta}) & \text{otherwise} \end{cases}$$

$\theta \leftarrow \theta - \alpha (Q(s, a | \theta) - y) \nabla_{\theta} Q(s, a | \theta)$ (or other opt. iteration)

if iter % 1000 = 0:

$\hat{\theta} = \theta$

← or other number
(iter is current global iteration id)

← target network weights

(check "rainbow" paper for more efficient/advanced techniques)

By sampling from M we remove high correlation between states in a single epoch and by using $\hat{\theta}$ we overcome overestimation problem.

Comparing to several hundreds or thousands epochs in standard DL, in RL we may have billions of them.

This is due to:

1. Don't have gradient of $r(s, a)$
2. Need to explore the environment, which is almost random walk.

Restrictions:

- ~~usable~~ only for small number of actions
- able to learn only deterministic policies

Policy gradient methods

$\tau = \{s_0, a_0, r_0, s_1, a_1, r_1, \dots\}$ - trajectory

$$p(\tau|\pi) = p(s_0) \prod_{t=0}^{\infty} \pi(a_t|s_t) \cdot p(s_{t+1}|s_t, a_t)$$

$$\text{Goal: } \mathbb{E}_{p(\tau|\pi)} (r_0 + \gamma r_1 + \gamma^2 r_2 + \dots) = \mathbb{E}_{p(\tau|\pi)} R_0 \rightarrow \max_{\pi} \gamma \in (0, 1)$$

Q-learning:

$$V^{\pi}(s) := \mathbb{E}(R_+ | s_t = s)$$

$$V^*(s) := \max_{\pi} V^{\pi}(s)$$

$$Q^{\pi}(s, a) := \mathbb{E}(R_+ | s_t = s, a_t = a)$$

$$Q^*(s, a) := \max_{\pi} Q^{\pi}(s, a)$$

$$\pi^*(a|s) = \operatorname{argmax}_a Q^*(s, a)$$

$$Q^*(s, a) = r(s, a) + \gamma \mathbb{E}_{p(s'|s, a)} \max_{a'} Q^*(s', a') \quad \forall s, a$$

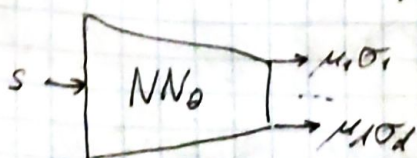
$$F(\theta) = \frac{1}{|S| \cdot |A|} \sum_{s, a} (Q(s, a | \theta) - \underbrace{r(s, a) - \gamma \mathbb{E}_{p(s'|s, a)} \max_{a'} Q(s', a' | \theta)}_{-y(s, a)})^2 \quad - \min_{\theta}$$

Saying the particular value of $Q(s, a)$ is much more sophisticated and unclear way than saying what would be the optimal action. So, learning $Q(s, a)$ may be more difficult problem, than learning $\pi(a|s)$.

The main difference from Q-learning is that we are going to directly learn $\pi(a|s, \theta)$.

$$\pi(a|s, \theta)$$

If $a \in \mathbb{R}^d$, we can produce pdf like in VAE:



$$\pi(a|s, \theta) := \mathcal{N}(a | \mu(s, \theta), \exp(\sigma(s, \theta)))$$

If $a \in \{1, \dots, K\}$

$$\pi(a|s, \theta) = \text{Softmax}(NN(s, \theta))$$

Since this way π is stochastic, we don't need to use exploration strategy (like ϵ -greedy).

Now the optimization problem is following:

$$F(\theta) = \mathbb{E}_{\pi(\theta)} R_0 \rightarrow \max_{\theta}$$

And we need to find $\nabla_{\theta} F$.

Reinforce (log-derivative trick)

$$\begin{aligned} \nabla_{\theta} \mathbb{E}_{\pi(\theta)} R_0 &= \nabla_{\theta} \int p(\tau|\theta) R_0 d\tau = \int \nabla_{\theta} p(\tau|\theta) R_0 d\tau = \\ &= \int p(\tau|\theta) \nabla_{\theta} \log p(\tau|\theta) R_0 d\tau = \mathbb{E}_{\pi(\theta)} \nabla_{\theta} \log p(\tau|\theta) R_0 \end{aligned}$$

Let's consider finite $\tau = \{s_0, a_0, r_0, \dots, s_T, a_T, r_T\}$

$$\mathbb{E}_{\pi(\theta)} \nabla_{\theta} \log p(\tau|\theta) \left(\sum_{t=0}^T \gamma^t r_t \right) =$$

$$\begin{aligned} &= \left| \begin{aligned} p(\tau|\theta) &= p(s_0) \prod_{t=0}^T \pi(a_t|s_t, \theta) p(s_{t+1}|s_t, a_t) \\ \nabla_{\theta} \log p(\tau|\theta) &= \sum_{t=0}^T \nabla_{\theta} \log \pi(a_t|s_t, \theta) \end{aligned} \right| \begin{aligned} &\text{we don't know, but it doesn't depend on } \theta \\ &\rightarrow = f(s_t, \theta) \text{ in case of determ. policies} \end{aligned} \end{aligned}$$

$$= \mathbb{E}_{\pi(\theta)} \left(\sum_{t=0}^T \nabla_{\theta} \log \pi(a_t|s_t, \theta) \right) \cdot \left(\sum_{t=0}^T \gamma^t r_t \right) =$$

$$= \mathbb{E}_{\pi(\theta)} \left(\sum_{t,s} \nabla_{\theta} \log \pi(a_t|s_t, \theta) \cdot \gamma^s r_s \right) =$$

$$= \sum_{t,s} \left(\mathbb{E}_{\pi(\theta)} \nabla_{\theta} \log \pi(a_t|s_t, \theta) \cdot \gamma^s r_s \right) \equiv$$

If $t > s$:

$$\begin{aligned} \mathbb{E}_{\pi(\theta)} &= \sum_{s_0, a_0, \dots, s_t} p(s_0, a_0, \dots, s_t | \theta) \gamma^s r_s \cdot \sum_{a_t} p(a_t | s_t, \theta) \cdot \nabla_{\theta} \log \pi(a_t | s_t, \theta) \cdot \sum_{s_{t+1}, a_{t+1}, \dots, a_T} p(s_{t+1}, \dots, a_T | s_t, a_t, \theta) \\ &\quad \left(\sum_{a_t} p(a_t | s_t, \theta) \cdot \nabla_{\theta} \log \pi(a_t | s_t, \theta) \right) \cdot \sum_{s_{t+1}, a_{t+1}, \dots, a_T} p(s_{t+1}, \dots, a_T | s_t, a_t, \theta) = 1 \end{aligned}$$

(*)

$$(*) = \nabla_{\theta} \sum_{a_t} \pi(a_t | s_t, \theta) = \nabla_{\theta} 1 = 0$$

If $t \leq S$: reward r_s will be dependent on a_t

$$\Rightarrow \sum_{t=0}^T \mathbb{E}_{\pi(\tau|\theta)} \nabla_{\theta} \log \pi(a_t | s_t, \theta) \sum_{s=t}^T \gamma^s r_s$$

To optimize we use stochastic estimation:

just play full episode, get τ and compute the gradient.

Reinforce scheme:

Initialize θ

for episodes from 1 to P :

Generate a trajectory $\tau = \{s_0, a_0, r_0, \dots, s_T, a_T, r_T\}$
using current θ

$R := 0, g := 0$

for $t = T, \dots, 0$:

$R \leftarrow r_t + \gamma R$

$g \leftarrow g + \gamma^t R \nabla_{\theta} \log \pi(a_t | s_t, \theta)$

$\theta \leftarrow \theta + \alpha g$ (or other opt. step)

Limitations:

1. Need to compute full trajectory for just one parameters (weights) updation.
2. Stochastic estimate of $\mathbb{E}_{\pi(\tau|\theta)}$ has a huge variance.

← explanation of the huge variance
Example: $\mathbb{E}_{\pi(\tau|\theta)} = \sum_{s_0, a_0, \dots, s_T} p(s_0, a_0, \dots, s_T | \theta)$.

$$\cdot \sum_{a_t} \pi(a_t | s_t, \theta) \cdot \nabla_{\theta} \log \pi(a_t | s_t, \theta).$$

$$\cdot \sum_{s_{t+1}, a_{t+1}, \dots, s_T, a_T} p(s_{t+1}, a_{t+1}, \dots, s_T, a_T | a_t, \theta) \cdot \sum_{s=t}^T \gamma^s r_s \quad \ominus$$

If we consider $\pi(a_t | s_t, \theta) = \pi(a_t | \theta)$ - independent from s_t . say, we have 2 actions and $\pi(a_t | \theta) = \frac{1}{2}$ for $a_t \in \{0, 1\}$.

Then

$$\ominus \sum_{a_t} \pi(a_t | \theta) \cdot \nabla_{\theta} \log \pi(a_t | \theta) \cdot \mathbb{E}_{p(s_{t+1}, \dots, a_T | a_t, \theta)} \sum_{s=t}^T \gamma^s r_s.$$

Say, r_s is big enough (in general, an arbitrary number) and we have

$$f_2 = \mathbb{E}_{p(s_{t+1}, \dots, a_T | a_t, \theta)} \sum_{s=t}^T \gamma^s r_s = \begin{cases} 100, & a_t = 0 \\ 101, & a_t = 1 \end{cases}$$

~~Then~~ and say $f_1 = \nabla_{\theta} \log \pi(a_t | \theta) = \begin{cases} 1 & \text{if } a_t = 0 \\ -1 & \text{if } a_t = 1 \end{cases}$

Then we have

$$\mathbb{E} f_1 \cdot f_2 = \frac{1}{2} \cdot 1 \cdot 100 + \frac{1}{2} \cdot (-1) \cdot 101 = -\frac{1}{2}$$

$$D f_1 \cdot f_2 = \mathbb{E} \left(f_1 f_2 + \frac{1}{2} \right)^2 = \frac{1}{2} \left(1 \cdot 100 + \frac{1}{2} \right)^2 + \frac{1}{2} \left((-1) \cdot 101 + \frac{1}{2} \right)^2 \approx 100^2 \gg 0$$

So, we introduce a baseline for r_s to reduce the variance. (Will discuss later)

Now, how to overcome the problem with sampling of full trajectories? We would like to use just one tuple (s_t, a_t, r_t) for updating θ .

Actor-Critic Approach

We sample full trajectory only to estimate the cumulative reward. So, let's combine Reinforce with DQN.

Let's use Q -function to estimate the cumulative reward.

Actor: $\pi(a | s, \theta)$

Critic: $Q^{\pi}(a, s)$ or $V^{\pi}(s)$

$$F(\theta) = \mathbb{E}_{p(\tau|\theta)} R_0 = \mathbb{E}_{p(s_0)} V^\pi(s_0) = F(\theta, s_0)$$

$$\nabla_\theta F(\theta, s_0) = \nabla_\theta \mathbb{E}_{p(s_0)} \cdot \mathbb{E}_{\pi(a_0|s_0, \theta)} Q^\pi(s_0, a_0) =$$

$$= \mathbb{E}_{p(s_0)} \sum_{a_0} \nabla_\theta (\pi(a_0|s_0, \theta) \cdot Q^\pi(s_0, a_0)) =$$

$$= \mathbb{E}_{p(s_0)} \sum_{a_0} (\nabla_\theta \pi(a_0|s_0, \theta) \cdot Q^\pi(s_0, a_0) + \pi(a_0|s_0, \theta) \cdot \nabla_\theta Q^\pi(s_0, a_0)) =$$

$$= \mathbb{E}_{p(s_0)} \sum_{a_0} (\nabla_\theta \pi(a_0|s_0, \theta) \cdot Q^\pi(s_0, a_0) + \pi(a_0|s_0, \theta) \nabla_\theta (r(s_0, a_0) + \gamma \mathbb{E}_{p(s'|s_0, a_0)} V^\pi(s')))) =$$

$$= \mathbb{E}_{p(s_0)} \sum_{a_0} (\nabla_\theta \pi(a_0|s_0, \theta) \cdot Q^\pi(s_0, a_0) + \pi(a_0|s_0, \theta) \cdot \gamma \underbrace{\mathbb{E}_{p(s'|s_0, a_0)} \nabla_\theta V^\pi(s')}_{= \nabla_\theta F(\theta, s')})$$

$$= \mathbb{E}_{p(s_0)} \sum_{a_0} (\nabla_\theta \pi(a_0|s_0, \theta) \cdot Q^\pi(s_0, a_0) + \gamma \pi(a_0|s_0, \theta) \cdot \mathbb{E}_{p(s_1|s_0, a_0)} \sum_{a_1} \nabla_\theta \pi(a_1|s_1, \theta) \cdot Q^\pi(s_1, a_1) + \pi(a_1|s_1, \theta) \cdot \gamma \cdot \mathbb{E}_{p(s_2|s_1, a_1)} \nabla_\theta V^\pi(s_2))) = \text{etc.}$$

$$= \sum_s \underbrace{\sum_{k=0}^{\infty} \gamma^k p(s_0 \rightarrow s, k, \pi)}_{\tilde{\mu}^\pi(s)} \cdot \sum_a \nabla_\theta \pi(a|s, \theta) \cdot Q^\pi(s, a) \quad \text{①}$$

$p(s_0 \rightarrow s, k, \pi)$ - prob. of obtaining state s after k steps starting from s_0 given policy π .

$\mu^\pi(s)$ is some p.d.f. over all states which is property of the environment for given policy π .

State s , which we observe at different time moments are just samples from $\mu^\pi(s)$.

$$\mu^\pi(s) = (1-\gamma) \cdot \tilde{\mu}^\pi(s)$$

$$\sum_s \mu^\pi(s) = \sum_s \sum_k \gamma^k p(s_0 \rightarrow s, k, \pi) = \sum_k \gamma^k \sum_s p(s_0 \rightarrow s, k, \pi) =$$

$$= \sum_{k=0}^{\infty} \gamma^k = \frac{1}{1-\gamma}$$

it's like weighted SGD

$$\text{② } (1-\gamma) \sum_s \mu^\pi(s) \sum_a \pi(a|s, \theta) \cdot \nabla_\theta \log \pi(a|s, \theta) \cdot Q^\pi(s, a)$$

$$\boxed{\nabla_\theta F(\theta) = (1-\gamma) \mathbb{E}_{\mu^\pi(s)} \mathbb{E}_{\pi(a|s, \theta)} \nabla_\theta \log \pi(a|s, \theta) \cdot Q^\pi(s, a)}$$

Now, ~~how~~ how we compute (where we get) $Q^\pi(a, s)$?

Let's use NN:

$$\begin{aligned} Q^\pi(a, s) &\approx Q(a, s | w) && \text{estimation for the} \\ \text{or } V^\pi(s) &\approx V(s | w) && \text{current policy } \pi, \text{ not} \\ &&& \text{the optimal like in} \\ &&& \text{Q-learning} \end{aligned}$$

$$Q^\pi(s, a) = r(s, a) + \gamma \mathbb{E}_{\pi(s'|s, a)} V^\pi(s')$$

$$V^\pi(s) = \mathbb{E}_{\pi(a|s)} (r(s, a) + \gamma \mathbb{E}_{\pi(s'|s, a)} V^\pi(s'))$$

↑
Bellman equation for V^π
 $y(s, a)$

$$F(w) = \frac{1}{|S|} \sum_s \left(V(s | w) - \underbrace{\mathbb{E}_{\pi(a|s)} (r(s, a) + \gamma \mathbb{E}_{\pi(s'|s, a)} V^\pi(s | w))}_{y(s, a)} \right)^2 \xrightarrow{\min_w}$$

$\sim (s, a, r, s')$

Now, we need to reduce the variance, discussed before:

Baseline:

$$\begin{aligned} \nabla_\theta F(\theta) &= (1-\gamma) \mathbb{E}_{\pi(s_0)} \mathbb{E}_{s \sim \mu^\pi} \mathbb{E}_{r(a|s, \theta)} \nabla_\theta \log \pi(a|s, \theta) \cdot \\ &\quad \cdot [Q^\pi(s, a) - B^\pi(s)] \end{aligned}$$

$$B^\pi(s) = \mathbb{E}_{\pi(a|s)} Q^\pi(s, a) = V^\pi(s)$$

Advantage: $A^\pi(s, a) := Q^\pi(s, a) - V^\pi(s)$

AAC (A2C, Advantage Actor-Critic) scheme

Initialize θ and w

for episodes from 1 to P :

for $t = 0, \dots, T$:

$$a_t \sim \pi(a_t | s_t, \theta)$$

get r_t and s_{t+1}

$$y = \begin{cases} r_t, & \text{if } s_t - \text{terminal state} \\ r_t + \gamma V(s_{t+1} | \hat{w}), & \text{otherwise} \end{cases}$$

$$A = \begin{cases} 0, & \text{if } s_t - \text{terminal state} \\ \underbrace{r_t + \gamma V(s_{t+1} | \hat{w}) - V(s_t | \hat{w})}_{\text{estimation of } Q(s_{t+1} | \hat{w})}, & \text{otherwise} \end{cases}$$

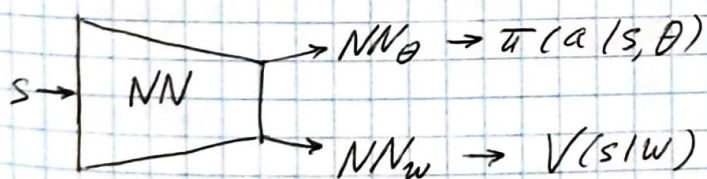
$$w \leftarrow w - \alpha_w (V(s_t | w) - y) \cdot \nabla_w V(s_t | w)$$

$$\theta \leftarrow \theta + \alpha_\theta A \cdot \nabla_\theta \log \pi(a_t | s_t, \theta)$$

if iter % $N = 0$:

$\hat{w} \leftarrow w$ - update each N iterations

Since $V(s | w)$ and $\pi(a | s, \theta)$ have the same inputs, it's better to couple them:



DQN - off-policy (you may use any samples (s, a, r, s') , including ones that obtained from expert's playing)

A2C - on-policy (you have to choose (s, a, r, s') according to your current policy) - inefficient learning.

SAC (soft actor critic), DDPG (Deep Deterministic Policy Gradient) - off-policy extensions to A2C