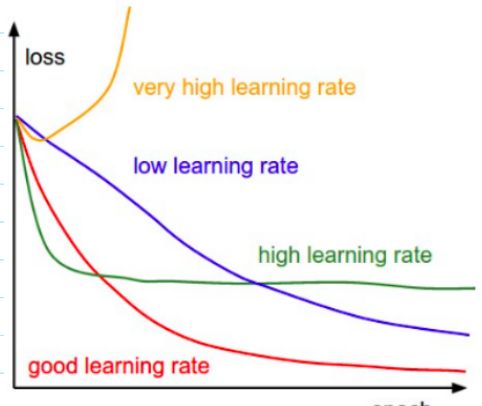
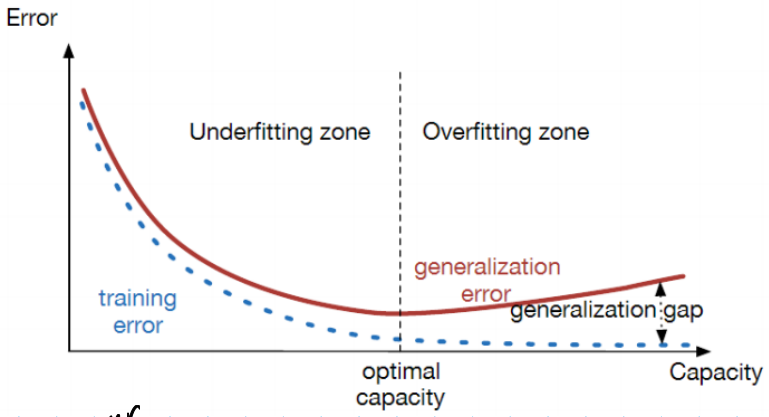
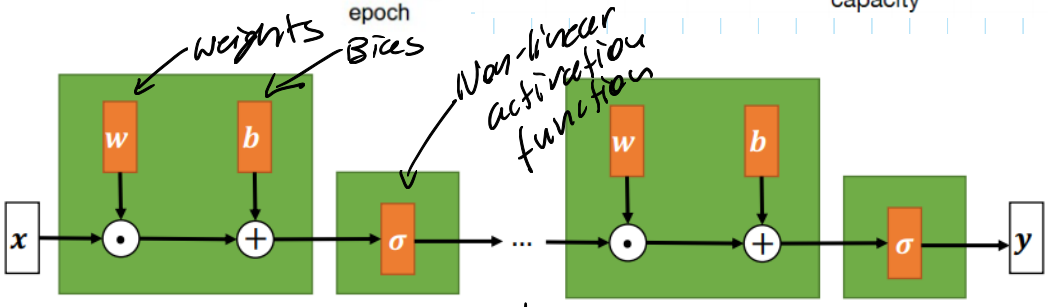
Machine Perception

- Maximum Likelihood : • $L(\theta) = \prod p(x_i|\theta)$
  • $\log(L(\theta)) = \sum \log p(x_i|\theta)$
  • Derive θ and set to 0

- Activation function
  - Must be non-linear, or the resulting function is just a linear mapping of the input
  - If network is linear, we can summarize network to only one block
  - Sigmoid function : – Logistic activation function
    – Binary classification
    – Output between $[0,1]$
  - Softmax function : – Multiclass
    – Summing all outputs = 1
  - ReLU : – Remove negative values







$\hookrightarrow x^{(k)} = \sigma(W^{(k)T} \cdot x^{(k-1)} + b^{(k)})$

- XOR example
  - Use neural network
  - Given two binary inputs, return XOR
  - Linear activation function can only linearly seperate input space

Assuming a linear function:

$$J(\Theta) = \frac{1}{(N=4)} \sum_{x \in X} (f^*(x^{(i)}) - f(x^{(i)}; \Theta))^2 \quad \text{with } f(\boldsymbol{x}; \boldsymbol{w}, b) = \boldsymbol{x}^T \boldsymbol{w} + b$$

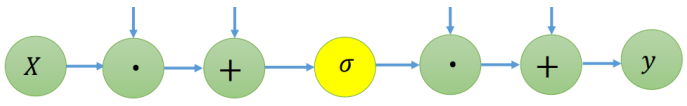
Solving via normal equation:

$$\boldsymbol{w} = \mathbf{0} \text{ and } b = \frac{1}{2}$$

1	0
0	1

  - With non-linear activation function like ReLu we transform input space

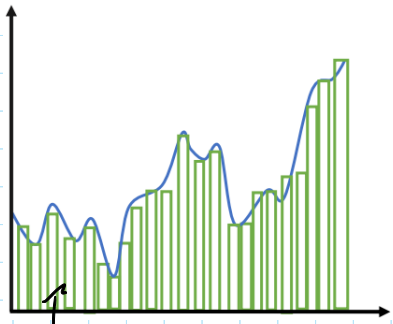
A feed-forward neural network with a single hidden layer and continuous non-linear activation function can approximate any continuous function with arbitrary precision.

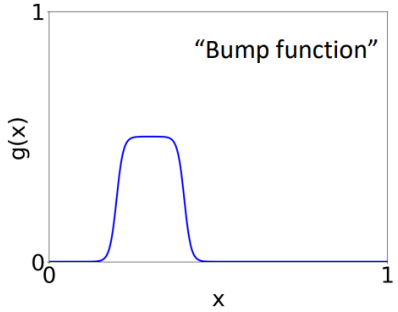


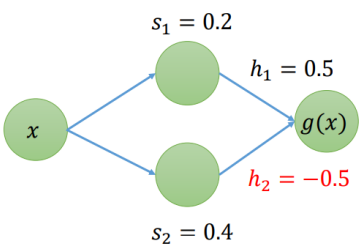
Transformation of input space do get seperability

- Universal Approximation theorem
  - Feed forward neural network
  - One single hidden layer
  - Non-linear activation function

} Able to approximate any continuous function with arbitrary precision







We are learning these green bumps

- Deeper networks work better than wider networks because we apply the activation function multiple times

- Backpropagation

- Chain Rule: $\frac{\partial f(g(x))}{\partial x} = \frac{\partial f(g(x))}{\partial g(x)} \cdot \frac{\partial g(x)}{\partial x}$

- new weight(i) = old weight(i) - Learning rate * $\left(\frac{\partial \text{error}}{\partial \text{weight}_i}\right)$

- Weighted input: $z = x \cdot w \Leftrightarrow z'(x) = w \;/\; z'(w) = x$

- ReLu: $R = \max(0, z) \Leftrightarrow R'(z) = \begin{cases} 0 & z < 0 \\ 1 & z > 1 \end{cases}$

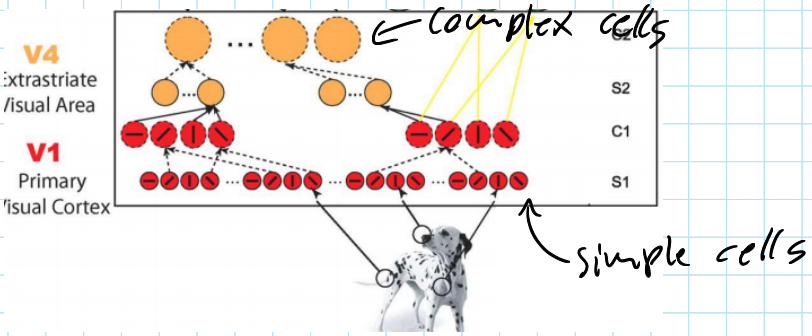
- Cost: $C = \frac{1}{2}(\hat{y} - y)^2 \Leftrightarrow C'(\hat{y}) = (\hat{y} - y)$

- If there are multiple paths to the weight, we must add all derived paths

- Convolutional Neural Network

- Simple cells: - Large cells
  - Come first in network

- Complex cells: - Combine simple cells output



- CNN: learn filters

Given a transform T, a function f, defining a translation or shift operation and input vectors $\boldsymbol{u}, \boldsymbol{v}$ scalars α, β

Definition: Transform T is *linear* if,

$$T(\alpha \boldsymbol{u} + \beta \boldsymbol{v}) = \alpha T(\boldsymbol{u}) + \beta T(\boldsymbol{v})$$

Definition: A transform T is *invariant* to f if, (shift operation, e.g. shift all by one to right)

$$T(f(\boldsymbol{u})) = T(\boldsymbol{u})$$

Definition: A transform T is *equivariant* to f if,

$$T(f(\boldsymbol{u})) = f(T(\boldsymbol{u}))$$

Any linear, shift-equivariant transform T can be written as a convolution

Correlation

$$I'(i,j) = \sum_{m=-k}^{k}\sum_{n=-k}^{k} K(m,n)I(i+m, j+n)$$

Doesn't really matter, because we learn the filters

Convolution

$$I'(i,j) = \sum_{m=-k}^{k}\sum_{n=-k}^{k} K(m,n)I(i-m, j-n)$$

$$= \sum_{m=-k}^{k}\sum_{n=-k}^{k} K(-m,-n)I(i+m, j+n)$$

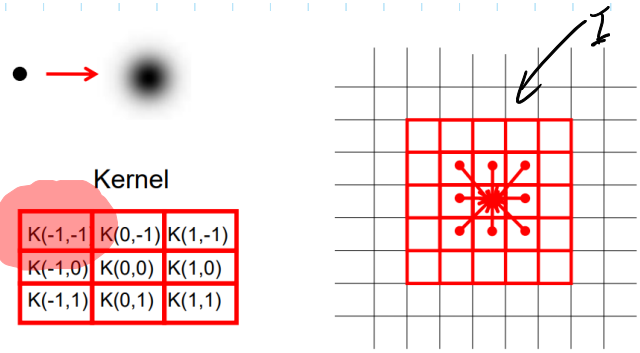
So if $K(i,j) = K(-i,-j)$, then Correlation == Convolution

$$O(i,j) = \sum_{m=-k}^{k}\sum_{n=-k}^{k} F(m,n)X(i-m, j-n)$$

Convolution $\left(\begin{bmatrix} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \\ X_{31} & X_{32} & X_{33} \end{bmatrix}, \begin{bmatrix} F_{11} & F_{12} \\ F_{21} & F_{22} \end{bmatrix} \right)$

$\Updownarrow$

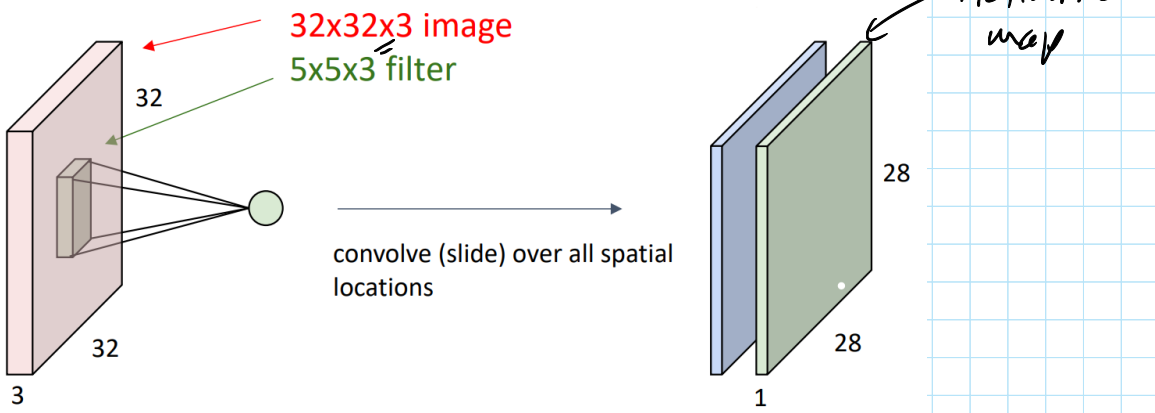
Correlation $\left(\begin{bmatrix} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \\ X_{31} & X_{32} & X_{33} \end{bmatrix}, \begin{bmatrix} F_{22} & F_{21} \\ F_{12} & F_{11} \end{bmatrix} \right)$



Kernel

K(-1,-1)	K(0,-1)	K(1,-1)
K(-1,0)	K(0,0)	K(1,0)
K(-1,1)	K(0,1)	K(1,1)

I' (x,y)=K(1,1)I(x-1,y-1) + K(0,1)I(x,y-1) + K(-1,1)I(x+1,y-1)
+ K(1,0)I(x-1,y) + K(0,0)I(x,y) + K(-1,0)I(x+1,y)
+ K(1,-1)I(x-1,y+1) + K(0,-1)I(x-1,y) + K(-1,-1)I(x+1,y+1)



- Pooling layer
  - Downscales image
  - operates on each activation map independently

Single depth slice

1	1	2	4
5	6	7	8
3	2	1	0
1	2	3	4

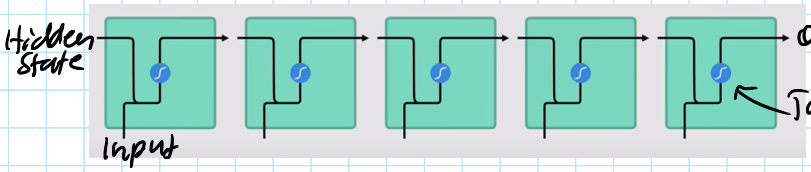
x, y

max pool with 2x2 filters and stride 2

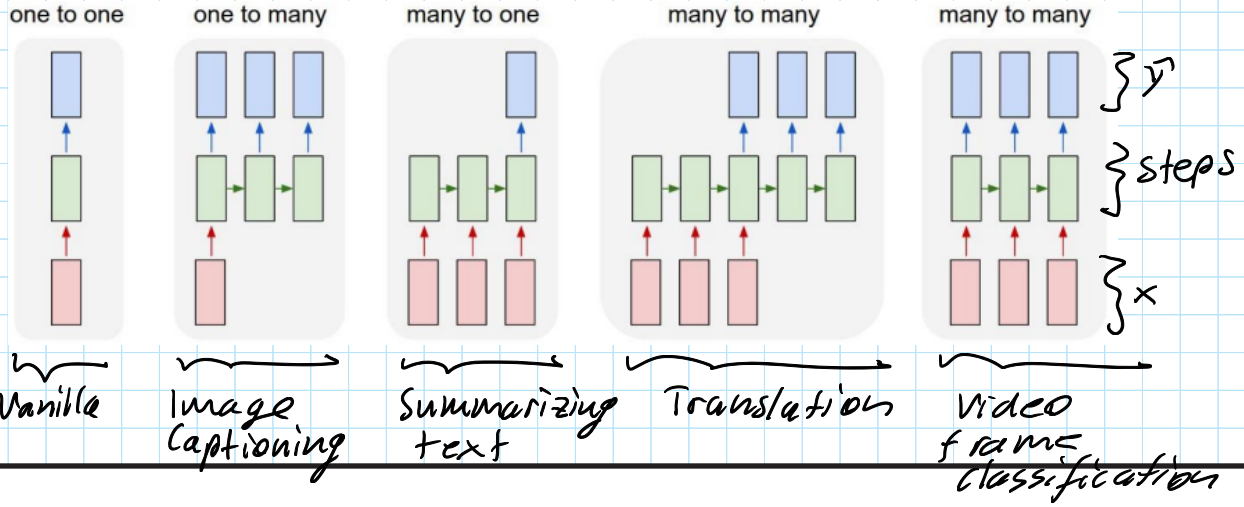
6	8
3	4

- Dense layer: last step before classification
  - Aggregate information

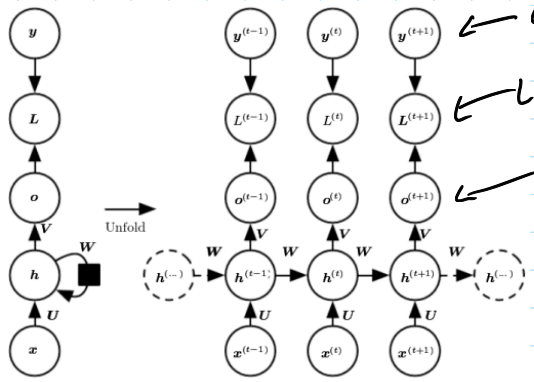
- Recurrent neural network



- Suffer from vanishing gradient problem
  - ↳ Gradient is getting smaller during backpropagation through RNN
    - ↳ If gradient is small, layer doesn't learn
    - ↳ If earlier layer can't learn, RNNs forget what it has seen in the long term
    - ↳ "short term memory effect"
- Hidden state flows from one step the next
- RNN work sequentially, one info at a time
- Hidden state is initialized as zero vector
- At each run, the hidden state is modified with the new input
- $h^t = \tanh(\underbrace{W_{hh} \cdot h^{t-1}}_{\text{Hidden state}} + \underbrace{W_{xh} \cdot x^t}_{\text{current input}} + \underbrace{b}_{\text{Bias}})$
- $\hat{y}^t = \text{softmax}(W_{hy} \cdot h^t + c \leftarrow \text{Bias})$
- Loss: $L = \sum_{t=1}^{T} L^{(t)} = \sum_{t=1}^{T} L(y^t, \hat{y}^t)$



- Parameters are shared between time-steps



← original y
← Loss
← output

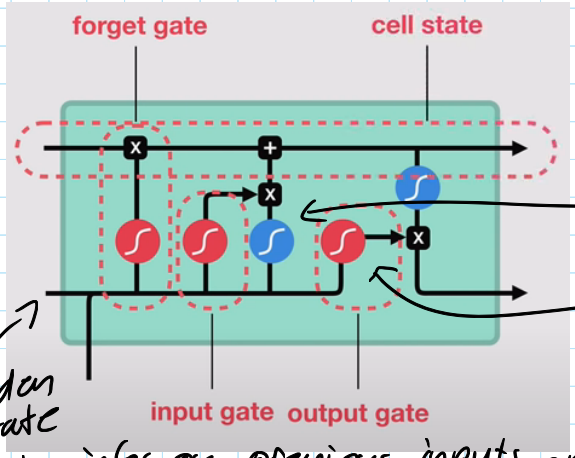
$$\frac{\partial L^t}{\partial W} = \sum_{k=1}^{t} \frac{\partial L^t}{\partial y^t}\frac{\partial y^t}{\partial h^t}\frac{\partial h^t}{\partial h^k}\frac{\partial h^k}{\partial W}$$

$$\frac{\partial h^t}{\partial h^k} = \prod_{i=k+1}^{t} \frac{\partial h^i}{\partial h^{i-1}}$$

↳ Exploding/vanishing gradients
(Gradient clipping) (LSTM or GRU)

- Long Short Term Memory (=LSTM)

- More evolved RNNs that doesn't suffer from vanishing gradient



← cell state transport relevant information (=memory)
← Regulation
← sigmoid ∈ [0,1]
↳ update or forget data
← Hidden state contains infos on previous inputs and is used for prediction

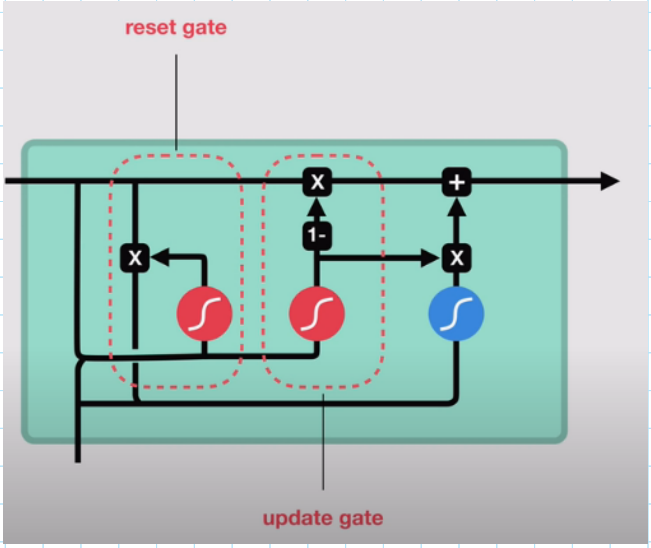
- Gates are also neural networks that learn, what info to keep or forget
- forget gate: what info should be forgotten from the previous cell state (0 ≙ forget)
- input gate: Decides how much new input we should take (0 ≙ not important)
- output gate: How much memory affects the output

forget-layer$(h^{t-1}+x^t)$ candidate-layer$(h^{t-1}+x^t)$

- c^t = forget gate output $\cdot\, c^{t-1}$ + input gate output $\cdot\, (h^{t-1}+x^t)$
- h^t = output gate output $\cdot \tanh(c^t)$

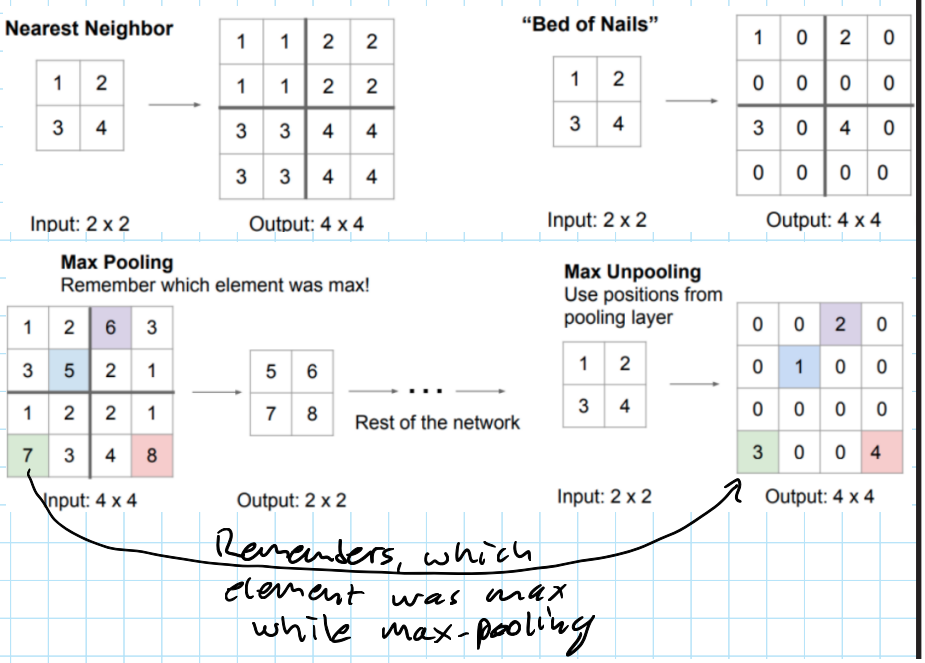
output-layer$(h^{t-1}+x^t)$

- Gated recurrent unit (=GRU)
  - Cell state is stored in hidden state
  - Fewer parameter than LSTM
  - It lacks output gate, because the memory is already in the hidden state

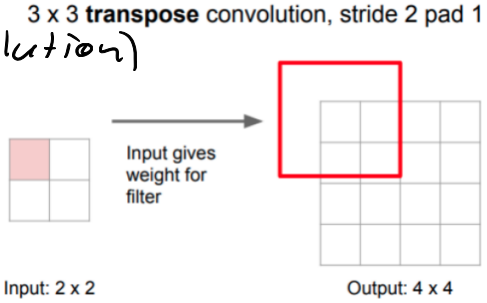


- The internal state of a standard RNN is entirely deterministic
  - They are often augmented with random latent variables to increase modelling capacity and to better capture uncertainty

- Object detection — Bounding box
  - As regression problem: Dog (x, y, w, h)
    - ↳ Need to test many positions and scales
  - As classification: 1. Find "blobby" image regions
    2. classify regions of interest
  - Semantic segmentation: · Label each pixel
    1. Extract patch
    2. Run through CNN
    - Get smaller output picture due to pooling
- Convolutions at original image resolution is too expensive
  - ↳ Downsampling → Upsampling → Output
    - Downsampling: · Strided convolution · Max-Pooling
    - Upsampling: · Unpooling
- Unpooling



- Learnable upsampling (= Deconvolution)
  - Use transpose convolutions
  - Learn filter



- Regularization
  - Any changes, which improve generalization error
  - Avoid overfitting
  - Constraining/regularizing coefficients towards zero to get a more flexible model
  - Manipulate data to avoid overfitting
    - Data standardization: better convergence

$$X_s = \frac{X - \mu}{\sigma}$$

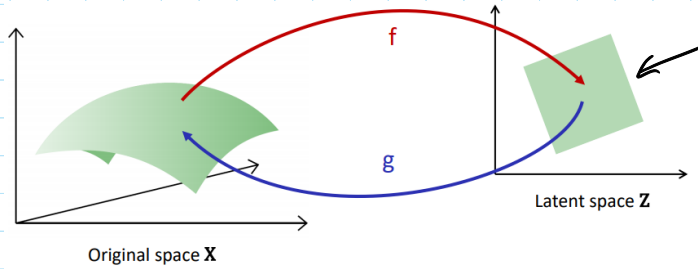
Where $\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$ and $\sigma = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \mu)^2}$

    - Data augmentation: generate synthetic data by transforming existing samples
      - add noise
  - Manipulate model to avoid overfitting
    - Penalize big weights:
      - $L(\theta) = L(\theta) + \alpha \Omega(\theta)$
      - L1: Lasso: sparse solution
      - L2: ridge: shrinkage
    - Dropout: Get rid of some connections

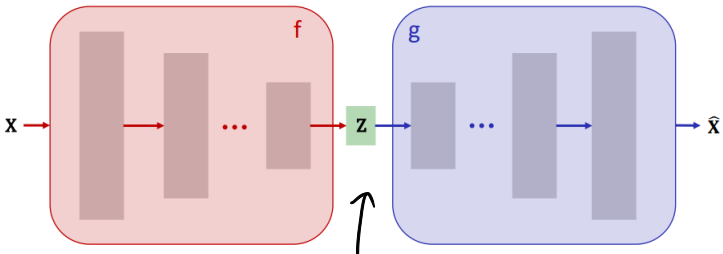
Dropout	Bagging
Models share weight	Models are independent
Trains partly only small percentage of its models	Trains all models until convergence

  - Pretraining: Train first on related task B with large dataset, finetune on task-specific dataset
  - Use pretrained network-architecture

- Generative Modelling
  - Given training data, generate new samples, drawn from training distribution
    - $p_{model}(x) \approx p_{data}(x)$
- Autoencoder
  - Transform high dimensional input to low-dim. so that each dimension describes Ausschlussmerkmal
  - Consists of encoder / decoder



smaller space



$x \approx \hat{x}$

decoder

Projected down

$$\hat{\theta}_f, \hat{\theta}_g = \underset{\theta_f, \theta_g}{\operatorname{argmin}} \sum_n^N \|x_n - \hat{x}_n\|^2$$

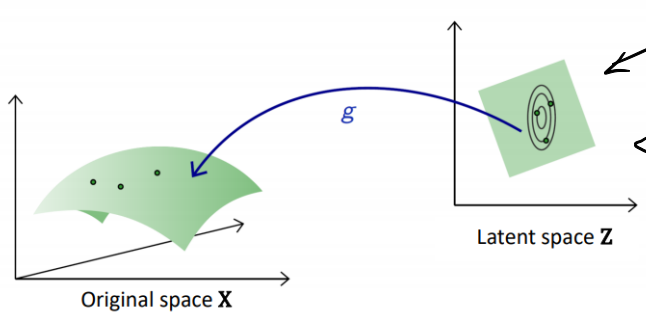
$$\hat{\theta}_f, \hat{\theta}_g = \underset{\theta_f, \theta_g}{\operatorname{argmin}} \sum_n^N \|x_n - g(f(x_n))\|^2$$

encoder

  - Used for human motion prediction
  - By introducing a simple density function (e.g. Gaussian) over the latent space z, the decoder g can generate new data

$$f(x) \sim N(\hat{\mu}, \hat{\sigma} I)$$

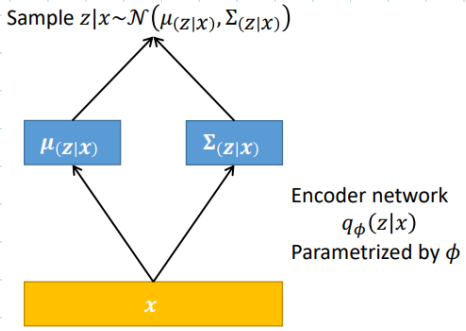
where $\hat{\mu}$ and $\hat{\sigma}$ are mean and standard deviation estimated from training data.

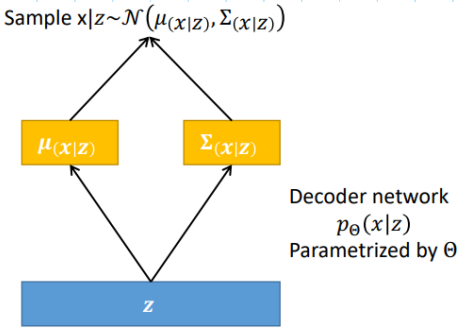


encoder is given distribution

Latent space is discrete, not continous, no easy interpolation

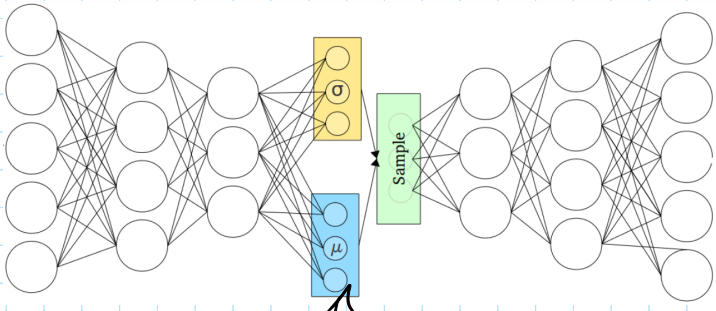
- Variational Autoencoder (=VAEs)
  - Probabilistic version of autoencoder => continous latent space
  - Generate data with natural variation
  - We want to estimate parameter θ^*
    - $p_\theta(x) = \int_z p_\theta(x|z) \cdot p_\theta(z) \cdot dz$
  - We use additional encoder network $q_\theta(z|x)$ to approximate $p_\theta(z|x)$
    - $p_\theta(z|x) = \frac{p_\theta(x|z) \cdot p_\theta(z)}{p_\theta(x)} \Rightarrow p_\theta(x) = \frac{p_\theta(x|z) \cdot p_\theta(z)}{p_\theta(z|x)}$ (prior)





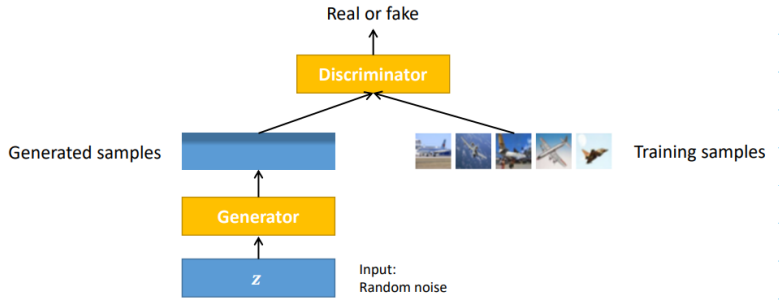
- Generates blurry image

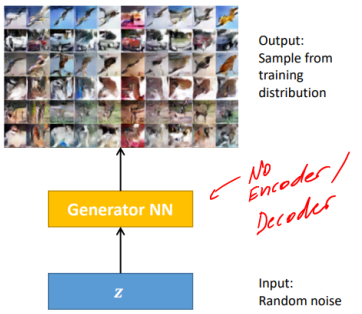
Used for generating Data, z is sampled from prior



Learned by encoder, sampled from by decoder

- Generative Adversarial Networks (=GANs)





No Encoder / Decoder

- Two networks play a two player game
- Generator: - tries to fool discriminator
  - $G: \mathbb{R}^D \to X$
- Discriminator: - tries to distinguish images
  - $D: X \to [0,1]$ (0 = fake)
  - If $D = 0{,}5$ everytime, generator has won

Assuming G is fixed, to train D given a set of real samples:

$$x^n \sim p_d, \qquad n = 1, \dots, N$$

We generate

$$z^n \sim \mathcal{N}(0,1), \qquad n = 1, \dots, N$$

and form a training dataset

$$\mathbb{D} = \{(x^{(1)}, 1), \dots, (x^{(n)}, 1), (G(z^{(1)}), 0), \dots, G(z^{(n)}), 0)\}$$

- Difficult to train: Nash-Equilibrium in two-player-game
  1. Make multiple steps in Generator
  2. Only make one step in Discriminator

Objective:

$$\min_{\Theta_g} \max_{\Theta_d} \mathbb{E}_{x \sim p_d(x)}[\log D_{\Theta_d}(x)] + \mathbb{E}_{z \sim p_z(z)}\left[\log\left(1 - D_{\Theta_d}\left(G_{\Theta_g}(z)\right)\right)\right]$$

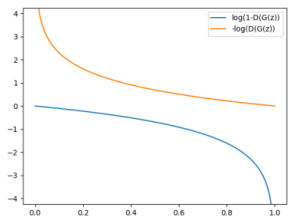
Alternate between:

1. Gradient *ascent* on D

$$\max_{\Theta_d} \mathbb{E}_{x \sim p_d(x)}[\log D_{\Theta_d}(x)] + \mathbb{E}_{z \sim p_z(z)}\left[\log\left(1 - D_{\Theta_d}\left(G_{\Theta_g}(z)\right)\right)\right]$$

2. Instead: gradient *ascent* on G:

$$\max_{\Theta_g} \mathbb{E}_{z \sim p_z(z)}\left[\log\left(D_{\Theta_d}\left(G_{\Theta_g}(z)\right)\right)\right]$$



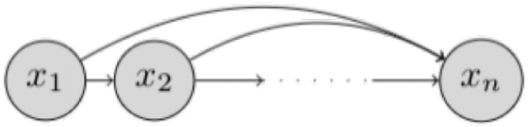
- Cycle GAN
  - Input -> Output -> input
  - Like in language, translate DE -> EN and then EN -> DE to check, if the meaning is preserved
- Mode collapse
  - Generator collapses and produces only limited varieties of samples
  - Generator focuses only on one mode

– Autoregressive Models

- Regression model uses data from same input variable at previous time steps, it is reffered to as autoregression
- Generative model models joint distribution $p(x, y)$
- Use x_1, x_2, x_3 to generate x_4
- Use x_2, x_3, x_4 to generate x_5

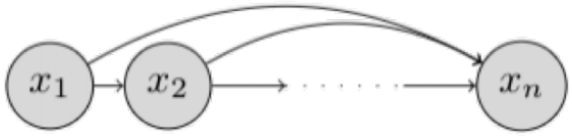
} Rinse n' repeat

$$p(x) = \prod_{1}^{n} p(x_i|x_1, \ldots, x_{i-1}) = \prod_{1}^{n} p(x_i|x_{<i})$$



} Attempt 1: Tabular approach

$$p_{\theta_i}(x_i|x_{<i}) = Ber(f(x_1, .. x_{i-1}))$$



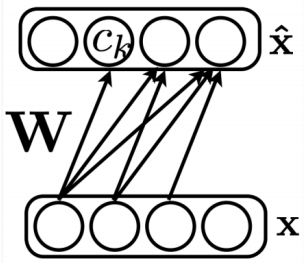
} Attempt 2

Fully Visible Belief Networks

$\hat{x}_i = p(x_i = 1|x_{<i})$

Modelled via logistic regression:

$f_i(x_1, x_2, \ldots, x_{i-1}) =$
$\sigma(\alpha_0^i + \alpha_1^i x_1 + \cdots + \alpha_i^i - 1 x_{i-1})$



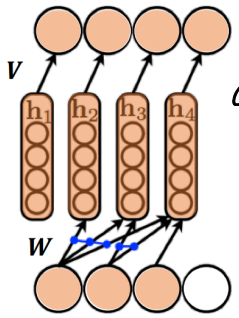
Neural Autoregressive Density Estimator (NADE)

autoencoder-like neural network to learn $p(x_i = 1|x_{<i})$:

$$\boldsymbol{h}^i = \sigma(\boldsymbol{b} + \boldsymbol{W}_{\cdot,<i}\, \boldsymbol{x}_{<i})$$
$$\hat{x}_i = \sigma(c_i + \boldsymbol{V}_{i,\cdot}\mathbf{h}^{(i)})$$

each conditional is modeled by the same neural network

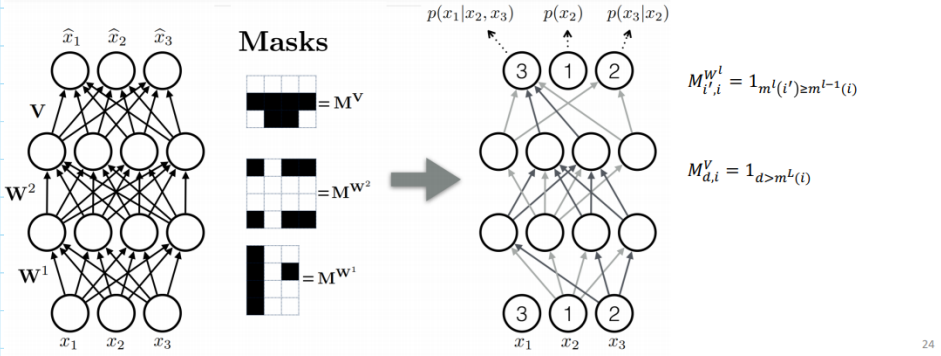
We can leverage the fact that:
$(\boldsymbol{b} + \boldsymbol{W}_{\cdot,<i+1}\, \boldsymbol{x}_{<i+1}) - (\boldsymbol{b} + \boldsymbol{W}_{\cdot,<i}\, \boldsymbol{x}_{<i}) = \boldsymbol{W}_{\cdot,i+1}\, \boldsymbol{x}_{i+1}$



only used for binary data

- Made (=Masked Autoencoder Distr. Estimator)
  - No computational path between output x_d and any of the input units must exist
  - Use masks to remove some connections

Constrain Autoencoder such that output can be used as conditionals $p(x_i|x_{<i})$



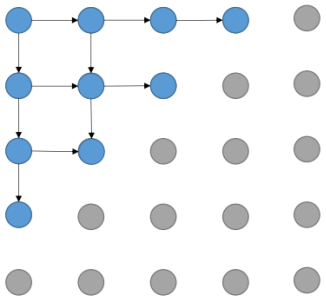
$$M^{W^l}_{i',i} = 1_{m^l(i') \geq m^{l-1}(i)}$$

$$M^{V}_{d,i} = 1_{d > m^L(i)}$$

PixelRNN [van der Oord et al. 2016]

Generate image pixels starting from corner

Dependency on previous pixels modeled using an RNN (LSTM)



Issue: Sequential generation is slow – due to explicit pixel dependencies

Faster, but still sequentially :(

PixelCNN [van der Oord et al. 2016b]

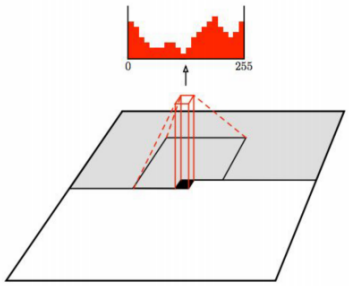
Still generate image pixels starting from corner

Dependency on previous pixels now modeled using a CNN over context region

Training maximizes likelihood of training images

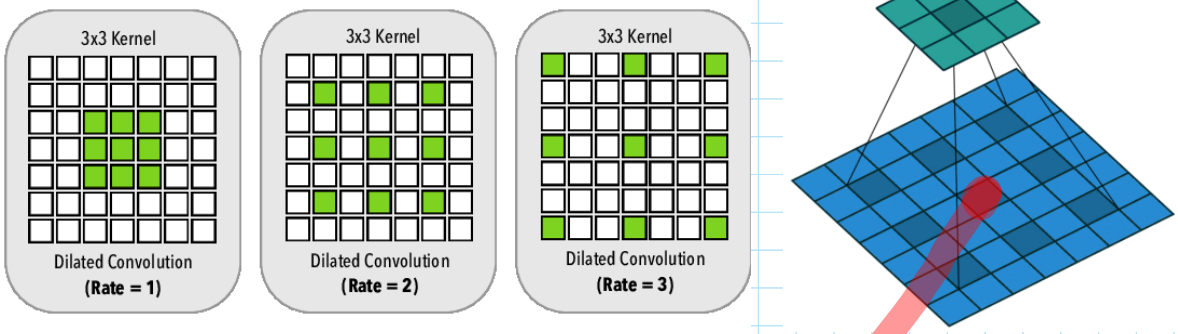
$$p(x) = \prod_{1}^{n} p(x_i|x_1, \ldots, x_{i-1})$$

Softmax loss over pixel values

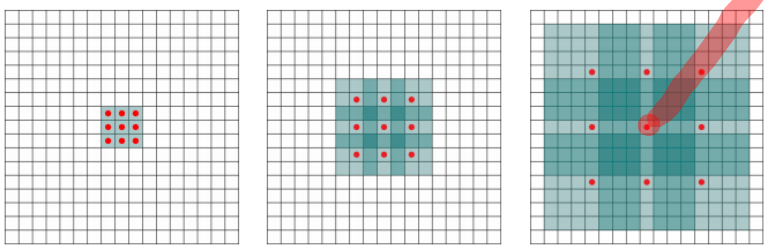


[image source: van der Oord et al.]

- Dilated Convolution



Receptive field size can increase exponentially with dilated layers

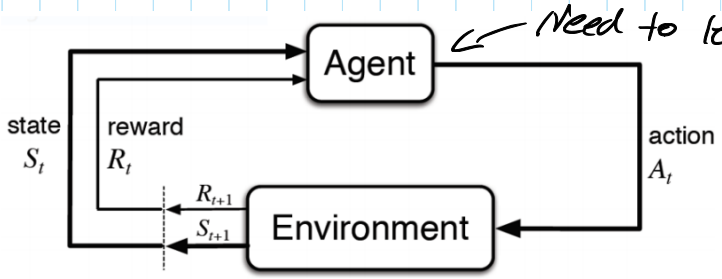


Layer 1 Layer 2 Layer 3

Red: Foot print of one dilated convolution
Shaded: Receptive field of convolution
Darker: Receptive field overlap between foot prints

- Reinforced Learning

- Learn how to act
- Maximize numerical award given



- We model environment as Markov Decision process

An MDP consists of $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathbb{P}, \gamma)$

- a set of states $\mathcal{S}$
- a set of actions $\mathcal{A}$
- a reward function $r: \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ (deterministic, can also be a distribution)
- a transition function $p: \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ (deterministic, can also be a distribution)
- initial state $s_0 \in \mathcal{S}$
- Discount factor: γ

- $V_\pi: S \to \mathbb{R}$ of a policy π returns the expected cumulative return for each state S when following policy π
- Dynamic Programming (=Tabular Learning => Tables)
  - We need whole transition matrix
    ↳ model of the world (=MDP)
  - Value iteration: 1. compute optimal values for each state V_*
    2. obtain policy π_* based on V_*
  - Policy iteration: 1. Initialize arbitrary policy π_+
    2. Repeat until policy doesn't improve
      2.1. evaluate value function V_+
      2.2. improve policy

- Temporal Difference Learning

For each step with action a from s to s' that we take with our policy, we compute the difference to our current estimate and update our value function:

$$\Delta V(s) = r(s,a) + \gamma V(s') - V(s)$$

$$V(s) \leftarrow V(s) + \alpha \Delta V(s)$$

← Anpassen des expected returns

- Random policy: · choose action randomly
  · Exploration, find unknown states
- Greedy policy: · choose best action
  · Can get stuck in local min.
  · Exploitation
- To get balance between exploration/exploitation we use epsilon-greedy-strategy
  ↳ $\varepsilon = 1$ ⇒ agent explore at start
  ↳ Through time, agent will get greedy

- Q-Learning (Q ≙ Quality value)

For each step with action a from s to s' that we take with our policy we compute the difference to our current estimate and update our value function like this:

discount factor

Data from exploration policy — Greedy policy to update the Q function

$$\Delta Q(S,A) = R_{t+1} + \gamma \max_a \{Q(S',a)\} - Q(S,A)$$

$$Q(S,A) \leftarrow Q(S,A) + \alpha \Delta Q(S,A)$$

↑ Get Δ

↑ Learning rate

Use this to update the fields

- Random policy for initialization
- Use initial Q-function and update it by exploration

immediate reward

$$Q(s,a) = r(s,a) + \gamma \max_a Q(s',a)$$

The Bellman Equation

↑ Reward from new state s'

The above equation states that the Q Value yielded from being at state s and selecting action a, is the *immediate reward* received, $r(s,a)$, plus the highest Q Value possible from state s' (which is the state we ended up in after taking action a from state s). We'll receive the highest Q Value from s' by choosing the action that maximizes the Q Value. We also introduce γ, usually called the *discount factor*, which controls the importance of longterm rewards versus the immediate one.

Not done yet — our greedy algorithm has a serious problem: if you keep selecting the same best-actions, you'll never try anything new, and you might miss a more rewarding approach just because you never tried it.

To solve this, we use an *ε-greedy* approach: for some $0 < \varepsilon < 1$, we choose the *greedy action* (using our table) with a probability $p = 1-\varepsilon$, or a *random action* with probability $p = \varepsilon$. We thus give the Agent a chance to *explore* new opportunities.

This algorithm is known as *Q Learning* (or *Q-Table*). Congratulations! you just learned your very first Reinforcement Learning algorithm!

· Deep Reinforced Learning (= Deep Q-Networks)
- we need to learn · Policy: $\pi: S_t \to A_t$
· Value: $V: S_t \to V(S_t)$)
- used if state space is large

Enter Deep Learning! We combine Q Learning and Deep Learning, which yields *Deep Q Networks*. The idea is simple: we'll replace the the Q Learning's table with a neural network that tries to approximate Q Values. It is usually referred to as the *approximator* or the *approximating function*, and denoted as $Q(s,a;\theta)$, where θ represents the trainable weights of the network.

Moving on: Training. In Reinforcement Learning, the training set is created as we go; we ask the Agent to try and select the best action using the current network — and we record the *state*, *action*, *reward* and the *next state* it ended up at. We decide on a batch size b, and after every time b new records were recorded, we select b records *at random (!!)* from the memory, and train the network. The memory buffers used are usually referred to as *Experience Replay*. Several types of such memories exist — one very common is a cyclic memory buffer. This makes sure the Agent keeps training over its new behavior rather than things that might no longer be relevant.

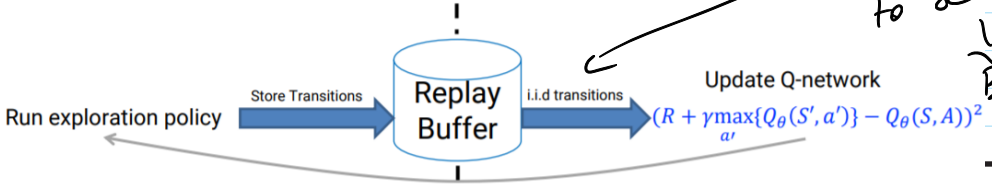
The Q-learning updates reduce to **stochastic gradient descent** on

$$Loss(\theta) = (R + \gamma \max\{Q_\theta(S', a')\} - Q_\theta(S, A))^2$$

How can we address this?

- Use a **replay buffer** to store generated samples
- During updates, randomly sample transitions from the buffer
- Since Q-learning is off-policy, we are allowed to use old samples

Samples need to be i.i.d. ⇒ Buffer



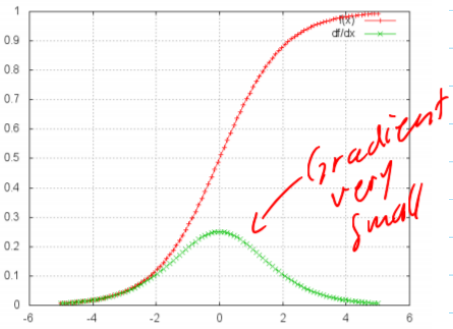
Sigmoid (Logistic)

+ Has a nice interpretation as the firing rate of a neuron: from not firing at all (0) to fully-saturated firing at an assumed maximum frequency (1).

+ Finite range (0,1): stable training.

- Saturates towards either end and large activations will get very small gradients.

- Not zero-centered. Always positive.



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

$$f'(x) = f(x)(1 - f(x))$$

Hyperbolic Tangent (tanH)

Scaled sigmoid function

$$tanh(x) = 2\sigma(2x) - 1$$

Like the sigmoid function

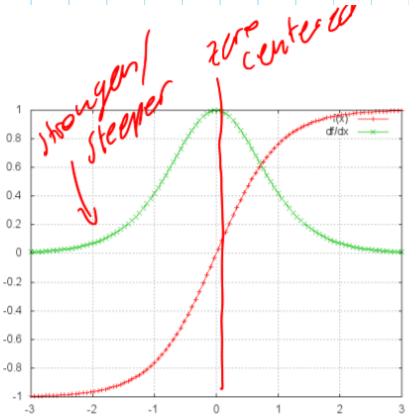
+ has finite range (-1,1),

- its activations saturate,

but unlike the sigmoid function

+ its output is zero-centered,

+ has stronger and steeper gradients.



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

$$f'(x) = 1 - f(x)^2$$

Rectified Linear Unit (ReLU)

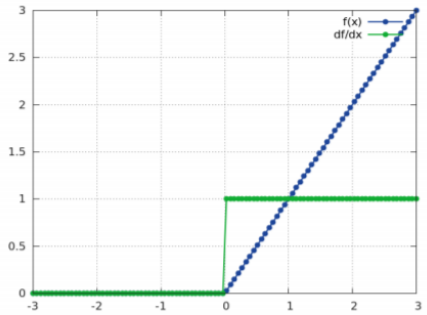
- range is [0, inf): it can blow up the activation and destabilize training.

+ range is [0, inf): It was presented to greatly accelerate the convergence.

+ inexpensive operation compared to sigmoid and tanh.

+ sparsity of the activations.

- “dying” ReLU problem: units with negative activations get no update.



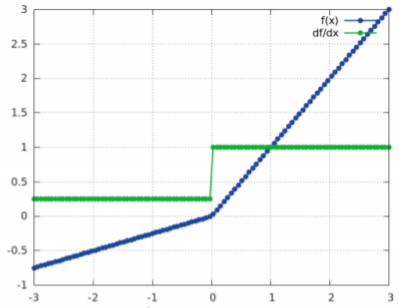
$$f(x) = max(0, x)$$

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

Leaky ReLU

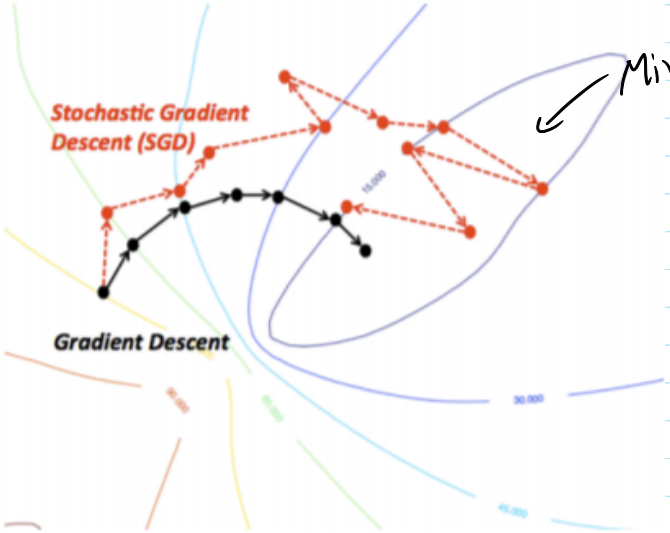
+ Attempts to solve “dying” ReLU by simply introducing a small negative slope for negative activations.

α is a small positive constant.



$$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$

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



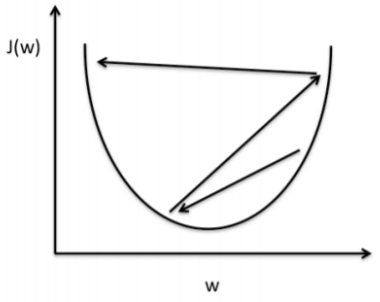
Optimization: AdaGrad (= Adaptive Gradient Descent)

Adagrad modifies the general learning rate η at each time step t for every parameter θ_i based on the past gradients that have been computed for θ_i

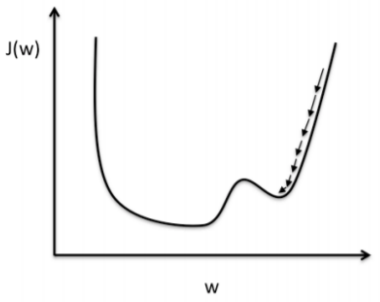
$$g_{t,i} = \nabla_\theta J(\theta_i)$$

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}$$

↳ Avoids overshooting



Large learning rate: Overshooting.



Small learning rate: Many iterations until convergence and trapping in local minima.