# Machine Learning

Early DL: perceptrons, neocognitron (like CNN), LeNet

Kullback-Leibler between distrs:

Terms: hyperparameters, train/val/test, curse of dimensionality, dataset normalization

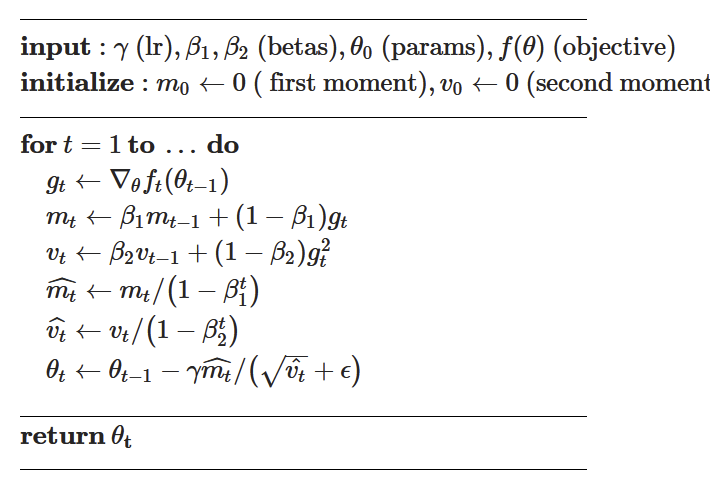
* Batchnorm: like dataset normalization, but without prior dataset statistics

Linear classifiers: (intuitively: learns one “template” per class); logistic (), softmax

* Losses: L2, CE (logistic) = -, hinge (SVM)

Optimization: Gradient descent (neg. gradient = dir. of steepest descent) (hyperparam: LR, weight init. method)

* Analytic gradient (exact, error-prone) vs. numeric (noisy, slow, easy - use to check analytic)
* Batch GD (expensive) vs SGD (GD on minibatches)
* GD strategies:
  + Momentum for GD:
  + AdaGrad: scale gradient element-wise by historic sum of squares (acts as per-element LR)
  + RMSProp/weighted AdaGrad: decay for past gradient
  + Adam: almost RMSProp + momentum (+bias correction)



* 2nd-order: invert Hessian -> use L-BFGS (only if doing full batch updates)

Regularization: L1/L2, dropout, batchnorm, cutout, mixup (encodes preference for weights), data augmentation

Linear classifiers: learn linear boundaries (feature transforms for nonlinear)

* Bag of Words: Extract random patches for image, cluster to form “codebook of visual words” (use as encoding for new model, e.g. SVM for classification)

# Neural Networks

Train via backprop - use the chain rule to derive gradients of loss with respect to every weight in network

* Can construct computational graph (indicating operations from inputs to outputs); fwd -> bwd pass
* Chain rule: downstream gradient upstream x local gradients
* Gradient flow: add distributes up grad to both eles; copy adds both up grad; multiplier multiplies up grad with multiplicand (a\*b -> a input multiplies by b); max gate passes through up grad for taken branch, 0 for others
* Backprop with vectors: look at N-d Jacobians instead

# Convolutional NN

Convolutional layer: slide kernel over image spatially (convolution as cross-correlation)

* Filter contains same # channels + same dim (besides # filters) as image; # filters = # output channels
* Shapes: , convolution shape:
* Later layers: resolution decreases, # channels increases (local -> complex features)

Stacked convolutional layers correspond to single larger convolutional layer (convolution - linear classifier)

* Padding to preserve feature map size; preserves input shape
* Each convolution adds to size of receptive field (total size: )
* Downsample with strided convolution (reduces # conv layers needed for receptive field)

Pooling: alternative way to downsample (hyperparam: kernel size, stride, pool function)

* Ex: max pool (take max within receptive field as output) -> gives invariance to small spatial shifts
* Pooling shape:

Batch norm: (normalize layer outputs across a batch to be zero mean, unit stdev)

* Testing: use running average from training (normalize each dim of input N-d vector x separately)
* Can learn D-dim scale, shift parameters for more info
* Batch-norm for CNNs: spatial batchnorm (batchnorm across each channel: take slices )
* Benefits: faster training/cvgnce, more robust to init., acts as regularization, can fold into conv layer during test
  + Provides robustness to internal covariate shift (change in distribution of network activations)
* Types of norm layers: batchnorm, layernorm/1D (for each input, compute mean/std across entire dim of input), instance/2D (for each input, compute mean/variance for each channel, separately)

Computing parameters: (rec: 4 bytes/float for memory)

* # params: total # weights + # biases [conv layer: , kernel size ]
* Floating point ops/FLOPs: number of output elems \* number of ops/elem [conv: ]

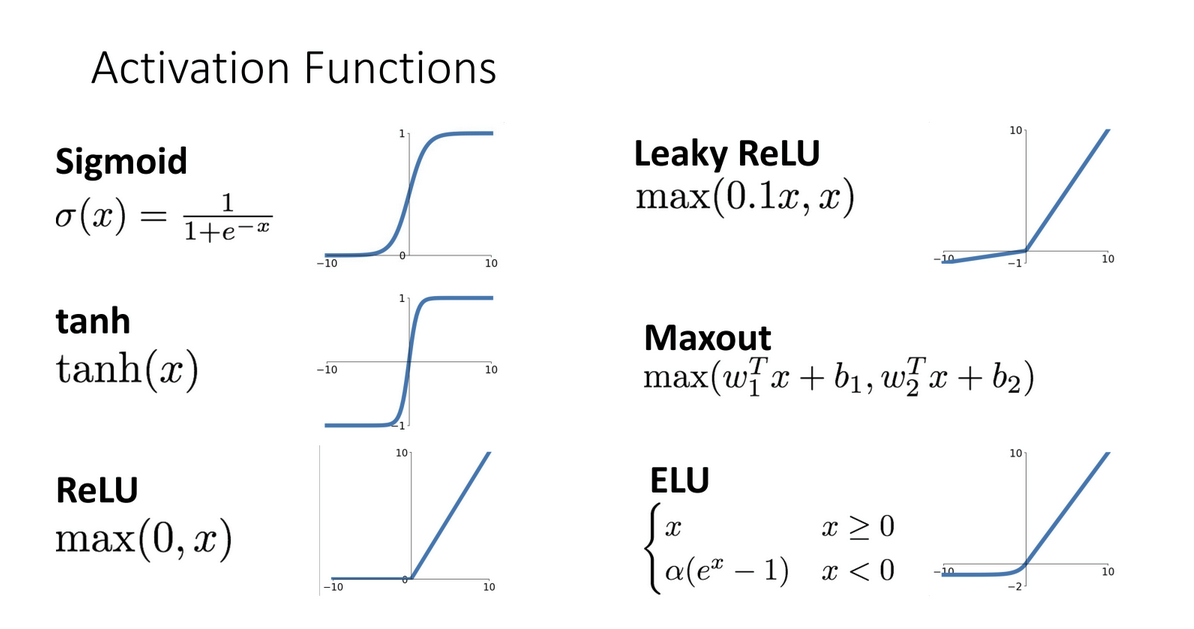
Modern architectures: (from LeNet)

* AlexNet: sigmoid -> ReLU, more training; (most mem in early convs; FLOPs in convs; params in FC)
  + ZFNet (changed kernel stride sizes), VGG (more regular design, much larger, high mem + FLOPs)
* GoogLeNet: stem network for early downsampling; glbl avg pool (take avg of each feature map as class score) instead of end FC (less params); Inception module: local unit with parallel branches (diff-sized convs)
  + Very eff; Hack: auxiliary classifiers (outputting classifications early) to help propagate gradients
* ResNets: residual connection to help deep networks learn identity (resblocks: 1st 0.5x size, 2x channels)
  + Eff + acc; Bottleneck block: replace 2 3x3 convs with 1x1->3x3->1x1 (more layers, cheaper)
  + Variations: (block design: ReLU after instead of before)
    - ResNeXt: G-many conv paths in parallel in res block, add w/ residual at end
      * Grouped conv: parallel conv layers work on subsets of channels (e.g. C / N), produce C\_out/N channels each -> parallelize for less RAM cost
    - Squeeze-and-Excite/SENet: glbl pool + 2x FC + within res block, \* with res output
* Densely connected NNs (each layer connected to all prev layers), MobileNets (replace conv with depthwise conv [group conv w/ # gps=# chnls] + ptwise 1x1 conv to reduce # params)

Neural architecture search: controller net outputs & trains child nets, updates self policy; give better archs over time

# Training NNs

Activation functions (want: nonlinear, diff’tble, no vanishing gradients):



* ReLU: well-behaved derivatives, no vanishing gradients for x>0, good in practice :); no 0 init/center :(
  + Leaky ReLU (no van grads), ELU (closer to 0-cen, neg sat regime compared to LReLU; more expensive)
  + Scaled ExponentialLU/SELU: for , if
    - Scaled version of ELU, better for deep NNs; self-normalizing (can train w/o batchnorm)
  + Gaussian ELU/GELU: (x by 0 or 1 randomly; larger -> more probability of x1, smaller -> more P of x0 [data-dependent dropout]); used in transformers
* Sigmoid: squashes numbers (0,1) [sigmoid, tanh not typically used except to squash]
* Tanh - squashes to [-1,1] & zero-centered :), but kills gradients :(
* Softplus

Weight initialization

* Bad: all 0 (all gradients equal, no symmetry breaking), small rand Gaussian (activations -> 0 for deep NN), large rand Gaussian (gradients saturate -> local gradients to 0, no learning)
* Xavier initialization for 0-center: stdev = (conv layers: ), ReLU: [Kaiming]
  + ResNets - variance grows with residual connection -> initialize first layer with Kaiming, later w/ 0

Regularization: stochastic depth (skip some blocks randomly), cutout, mixup; dropout for large FC layers

Learning rate schedules: step (reduce at fixed points), cosine , linear, inverse sqrt

Early stopping, or record best iter. (train w/ train, val w/ val) & repeat to iter. with train + val together

Choosing hyperparameters: grid search (log-linear scale, e.g.), random search (log-uniform on an interval)

* Random search: good if one parameter known more important (covers more total values)

Model ensembles for slightly better performance

Transfer learning: use pretrained CNN, remove FC layers, replace with new MLP head & retrain

* Advanced: lower learning rate, freeze lower layers; allows for reusing feature extraction
* Alt: instead of training feature extractor on large labeled, train on large unlabeled (unsupervised)

# Understanding Neural Networks

Can visualize convolutional filters in CNN (mainly lower layers)

* Multimodal neurons - some neurons in CNNs become object detectors for specific object classes

Looking at features - collect feature vectors from running on many images, k-NN/PCA to compare in feature space

Annotating interpretation of images - dissect networks to find “interpretable units” corresp. to each label

Maximally-activating patches: given layer + channel, run many images & find patches maximizing that channel val

Saliency via occlusion (mask part of image before fwd pass, see output change), backprop (compute gradient of unnormalized class scores w.r.t. each pixel); can also use saliency maps as form of unsupervised segmentation

Class-activation mapping/CAM: rather than summing all etnries (across image) in final FC layer to produce a class score, output matrix of final FC values as that class’s CAM (can also use as weakly-supervised object detection)

* Issue: only works for last conv (not GAP) -> gradient-weighted CAM/Grad-CAM: pick any layer’s activations, compute grad of class score w.r.t. activations, use GAP on gradients to get weights -> ReLU to find CAM

Visualizing CNN features: gradient ascent compute synthetic image maximally activating neuron

* Start with zero image, keep fwd pass & step input image in dir. of positive gradient (L2 + blur, clip also)
* Can use to visualizes; alt: train generator net (prior) for synthetic, followed by CNN (backprop CNN & gen net)
* Adversarial examples: given an arbitrary image, start with arbitrary class & grad asc to fool network
* DeepDream - try to amplify neuron activations at some layer in output -> trippy output

Feature inversion - given CNN feature vector, find new “natural-looking” image w/ similar feature vector (via regulari.)

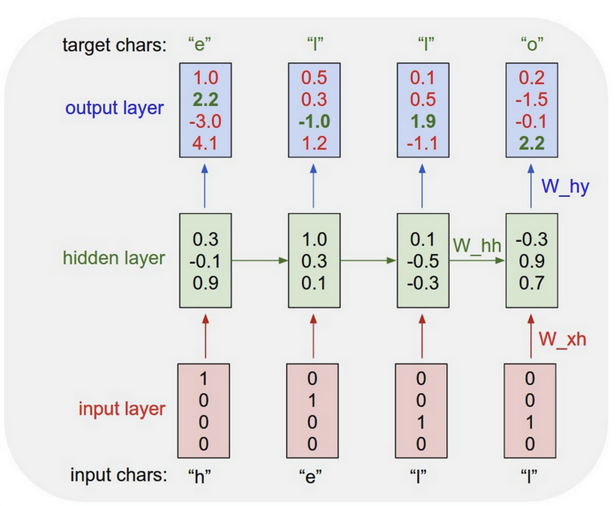
Texture synthesis: given sample patch of some texture, try to generate bigger image with same texture

* Reshape features C x H x W -> C x HW & find Gram matrix; neural texture synthesis: pretrained CNN fwd pass & find Gram matrices on every layer; initialize gen. Image from random noise, fwd pass, find Gram matrices
  + Compute weighted sum of L2 dist. between Gram matrices, backprop for gradient, make step
* Style transfer: texture transfer + Gram reconstruction (matches features from image 1, Gram Ms from image 2)
  + Fast neural style transfer - train an NN to copy style transfer (one net/style or condi. instance norm.)

# RNNs

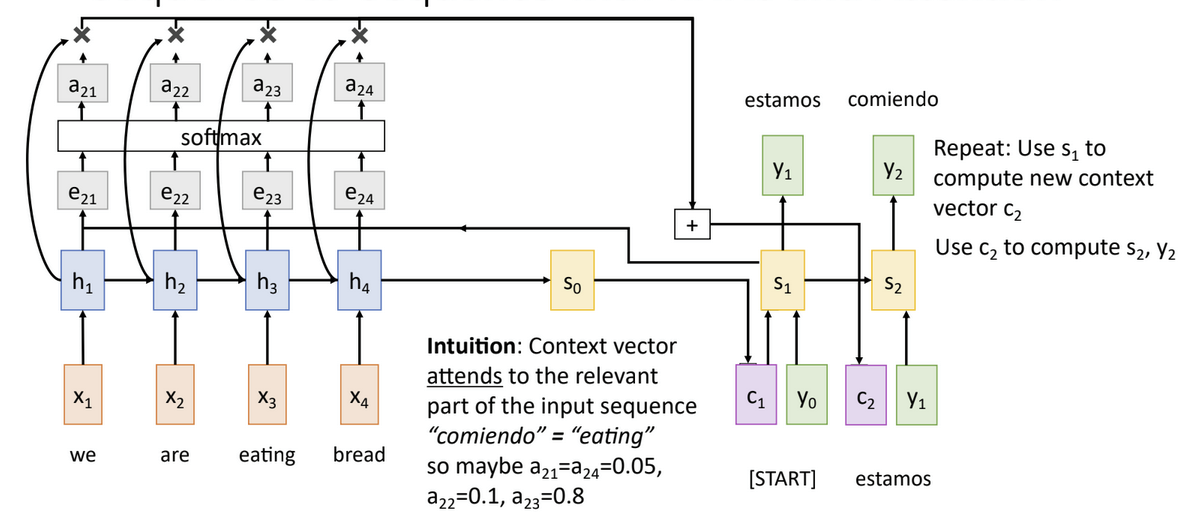
RNNs (vanilla): store state - single hidden vector , new state product of old state & input vector

* Language modeling: given characters 1, 2, …, t-1, char at time t is function of hidden state + preceding char
  + At test time: generate new text; sample chars 1-by-1 and feed back to model as next preceding char
  + Image captioning: use CNN feature vector as initial hidden state for RNN



RNNs w/ Attention (sequence-to-sequence) - decoder uses context vector , preceding char , preceding decoder state to update decoder state: & sample from decoder

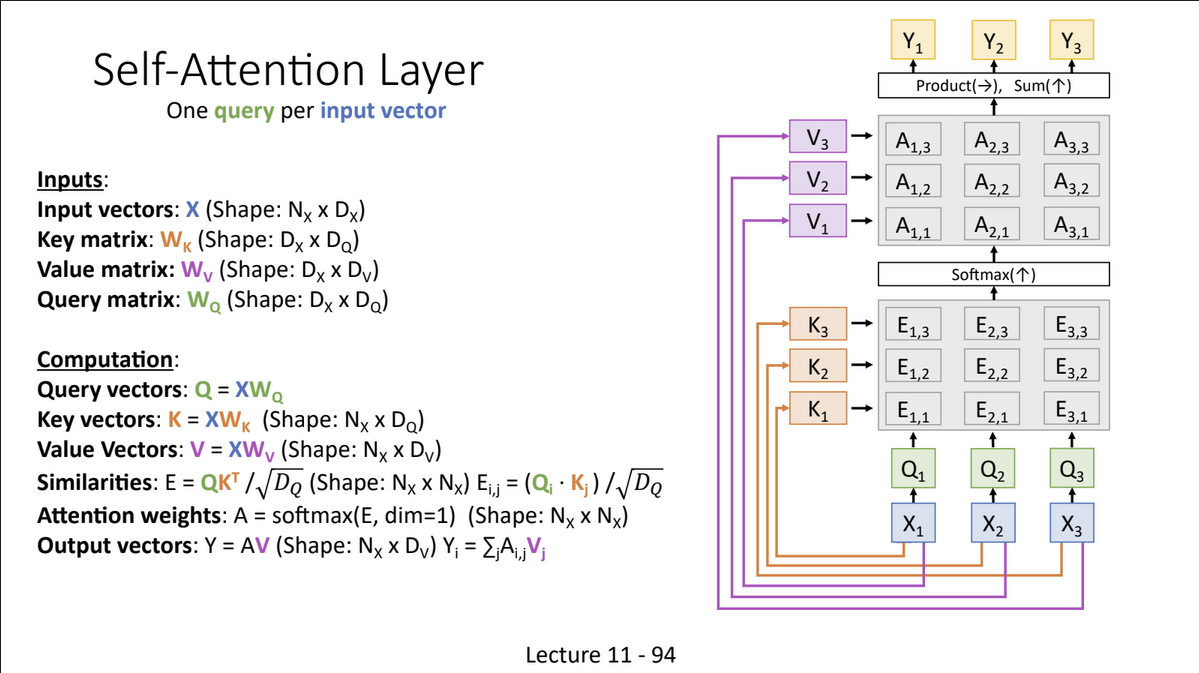
* Context vector: at each step, use decoder state to compute alignment scores (using MLP ; from final hidden state ), softmax alignments for att weights, context vector as linear combination of hidden states -> use as input to decoder (’s don’t need any order)



* Image captioning: use CNN to compute grid of features -> use to compute context vector

Attention variations: (query - decoder state, input - hidden states)

* Attention: queries , inputs -> scaled dot product: similarities
* Key-val attention: key, value matrices , -> sims: prod of , ; output
* Self-attention: uses query matrix to compute query vectors , in addn. to key/value vectors
* Masked self-att: zeroes out similarities with keys “further ahead” of current query vec
* Multihead self-att: H parallel self-att heads; concatenate outputs at end



# Transformers

Transformer block: self-attention w/ residual -> layernorm -> MLP (indep., on each vector) -> layernorm -> output

Transformers (sequences of transformer blocks) -> highly scalable, parallelizable (vectors only interact in self-atts)

* Multihead attention: runs parallel attentions, concatenates -> MLP on output to transform dimension
* Cross-att: instead of Q/K/V from input, Q from decoder & K/V from encoder (D: target, E: source)

Attention for vision

* Add attention to existing CNNs -> still a CNN
* Replace conv with local attention around receptive field -> not much improvement
* Standard transformer on pixels -> very memory-intensive (R x R image -> elements/attention matrix)

Vision Transformer (ViT): divide image into smaller flattened patches (concatenate with learned pos. embedding)

* Pass image patches as input to transformer; add special learned classification token (same dim as image patches) -> output vector corresp. to classification token is vector of class scores; no conv layers needed
* Scales better than ResNet for large datasets, + faster to train (but worse on small datasets)
* Distillation: Train teacher model from GT -> train student to match teacher predictions

Hierarchical ViT/Swin transformer: splitimage into patches (e.g. H/4x W/4) -> halve patch dimension between blocks

* Merging step: concatenate groups of 2x2 & linear project to half number of channels
  + Gives hierarchical structure (similar to CNNs)
* Issue: matrices big for earlier layers -> window attention: rather than all tokens attending all tokens, divide attention matrix into smaller M x M windows & only compute attention within each window linear in M)
  + Swin transformer - instead of pos. embeddings (ViT), encodes relative position btwn patches + bias
* Issue: no interaction between windows -> alternate between normal, shifted windows in each block

MLP-Mixer: use MLP to mix across tokens (replacing self-attention) -> all-MLP architecture

Object detection with transformers

* DETR (simple pipeline): directly output set of bounding boxes from transformer; use transformer to encode & another to decode, generate output vectors; use FFNs to generate prediction (no object, or class + box)
* Diffusion models with Transfs (DiT): replaces latent diffusion U-Net backbone w/ transformer on latent patches

# Object Detection

Crop (sliding window) & CNN for classification on window (bounding box: (x, y, w, h); L2 loss)

R-CNN: On proposed regions, transform/resize -> forward through CNN with classification

* CNN: predict class & bounding box correction/transform (select subset of detections to output)
* Transform: RoI pool (snap onto grid cells) vs RoI align (bilinear interp. - look at 4 nearest points x/y)

Fast R-CNN: Run CNN first -> on features: region proposal + transform + per-region CNN

Faster R-CNN: Learnable region proposal network from feature map (backbone + RPN -> per-region CNN)

* At each point in feature map: take K anchor boxes, predict if contains object (+ object box, if so)

Single-stage detection: instead of RPN output is/isn’t object, predict as one of C classes (or BG) + BBox TF

Indices: IoU/Jaccard

* NMS: select next highest-scoring output box & eliminate worse boxes with significant IoU
* Mean Avg Prec/mAP: for each detection, if matches GT w/ IoU>0.5, mark (+) & eliminate GT box; else, mark (-) [avg prec: area under prec vs recall curve] -> mAP: average across all categories
  + Prec: TP / (TP + FP); recall: TP / (TP + FN); COCO mAP: avg across multiple IoU thresholds

# Semantic Segmentation

U-Net: downsample (maxpool, strided conv) -> upsample (unpool, transpose conv)

* Unpooling: simple (only fill top-left), NNghb, interp, “max unpooling” (remember maxpool pos)
* Transpose conv: map single pixel in input to larger kernel (e.g. 3x3) in output
  + Move one input in input -> two in output, e.g.; sum where kernels overlap (int: x by A-1)

Other: dilated/astrous convolution (spread-out kernel pixels), pyramid structures; Cityscapes/ADE20K

# Instance Segmentation

Mask R-CNN: Faster R-CNN for object detection + add segmentation mask prediction to outputs

Other tasks: panoptic (inst + sem seg), keypoint (Mask R-CNN + 1-pixel keypoint mask), Mesh R-CNN (mesh head)

# Generative Models

Supervised/discriminative (learn p(y|x) ) vs. unsupervised/generative (learn + sample from p(x))

* Discriminative: given images, labels compete for prob. mass vs gen: images for mass (also: cond. gen.)

Generative models:

* Explicit density: autoregressive (tractable -exact p(x)) vs approximate (VAE - variational lower bound, diffusion)
* Implicit density: GANs (don’t compute p(x), only sample from it)

Autoregressive: want to learn -> solve for dataset

* Idea: for multi-part inputs , use conditional probs
* PixelRNN: generates pixels 1-by-1 from upper left (RNN: computes hidden state based on RGB of pixels directly to left and above; for each pixel, predict RGB separately & softmax) -> issue: slow due to sequential
* PixelCNN: CNN instead of LSTM for dependency on prev. pixels (parallelizable conv, but still slow sequential)
* Pros: explicitly computes , good samples; cons: slow due to sequential steps

Autoencoders (unsupervised for feature extraction): train model as encoder & decoder (decoder attempts to reconstruct original input data from encoded features; e.g. encoder conv -> decoder tconv); encoder is autoenc

* Can use encoder to initializer supervised models [encoder: high -> low dim, decoder: low -> orig. dim]

Variational autoencoders/VAE: attempts to learn latent features from input data -> sample from model to generate

* Assume training data x1,...,xN from latent rep. (latent features; e.g. attributes, orientation, etc.)
* Idea: Learn ; via decoder network (from Gaussian prior, e.g.); via trained encoder [want to approximate ]
* Network: encoder network takes input , gives distribution over latents [learns ]
  + Decoder network takes latent , gives distribution over [learns , ]; train jointly
  + Tries to maximize varia. LB ELBO on likelihood:
* FC VAE: each encoder, decoder is one initial linear layer + two parallel linear layers for ,
* Training VAE: run input data through encoder to get distr over latents, sample latents from encoder output -> run latent through decoder to get distr over data (want orig input data to have high probability)
  + Sampling: sample from prior, use decoder to get distr over & sample [modify -> change attrs]
* Pros: mathematically principled, rich latent space; cons: doesn’t find [only maximizes LB], bad quality

Vector-quantized VAE (VQ-VAE): VAE-like encoder generates latent space; autoregressive on latent space as decoder

* Train VAE encoder-decoder to generate grids of latent codes from input (continuous latent sp -> disc distr via vector quant.); PixelCNN to sample from latents (improves in VAE image quality)

Generative adversarial networks (GANs): train generator G to fool discriminator D

* Assume have data from -> introduce latent variable with simple prior , sample to G and train generator to sample, fool discriminator D (D a classifier - real/fake) [global min: ]
* Train D, G jointly via minimax [] (note: difficult, unstable training/loss)
  + Train G to minimize to avoid vanishing gradients
* Layer-wise stochast. (StyleGAN): GAN only initial random latent input -> fixed input + new rand latent/layer
* GANs: latent space encodes semantic info, continuous (can traverse, identify subspaces)

Image-to-image transl: can use GANs to translate images between diff domains (generator takes input image as input instead of random noise; discriminator takes input paired image, generator-output image)

* Non-paired translation: CycleGAN - take two sets of images (one in each domain) w/o pairing, use cycle reconstruction loss - minimize reconstruction error from converting from one domain to other & back

Diffusion models

Training: Forward process (at each step , add some noise sampled from standard normal; ) -> denoising proc. (use noise from encoding process to denoise; )

* Markov chain process - each step depends only on output of previous step
  + Hyperparams: noise schedule ; at each step of diffusion proc., moves toward
* Denoising: have individual mappings mapping noise to (via learned NN - inputs image, time embedding) -> want to learn
* In practice: use diffusion encoder/decoder as encoder/decoder in U-Net; from noise , use U-Net as denoising network for each step [ many U-Nets; from noisy image + time enc, preds. noise]
  + After training: sample white noise, pass to decoder to generate an image

Latent diffusion: use encoder/decoder to move from pixel to latent space (like VAE); diffusion process performed within latent space, encoder/decoder convert input, output to, from pixel space to latent space (faster + more eff)

* Extensions: stable diffusion, control (ControlNet: input condition image & text prompt; incorporate image encodings into diffusion decoder stages)