

Stanley Wei

CS 163

Zhou - F24

Contents

| | |
|--|-----------|
| Contents..... | 1 |
| Background..... | 2 |
| Machine Learning..... | 2 |
| Neural Networks..... | 6 |
| Backpropagation..... | 7 |
| Convolutional Neural Networks..... | 8 |
| Convolution & Normalization Layers..... | 8 |
| Modern CNN Architectures..... | 12 |
| ResNet..... | 12 |
| Training Neural Networks..... | 15 |
| Initialization..... | 15 |
| Training..... | 18 |
| Choosing Hyperparameters..... | 19 |
| After Training..... | 20 |
| Transfer Learning..... | 20 |
| Visualizing & Understanding Neural Networks..... | 21 |
| RNNs & Attention..... | 25 |
| Attention..... | 26 |
| Transformers..... | 29 |
| Computer Vision..... | 33 |
| Object Detection..... | 33 |
| Semantic Segmentation..... | 37 |
| Instance Segmentation..... | 39 |
| Generative Models..... | 41 |
| Autoregressive Models..... | 43 |
| Variational Autoencoders..... | 44 |
| Generative Adversarial Networks (GANs)..... | 47 |
| Diffusion Models..... | 49 |
| Trends in CV..... | 51 |

Background

Machine Learning

Linear classifiers: Given input x /weights W , outputs $f(x, W)=Wx+b$

- Predict a vector (corresponding to class scores)
 - Bias trick to incorporate biases as last column of weights
- Is a single-layer NN (perceptron); outputs linear predictions
 - Separates space of inputs into different decision regions
 - Cannot predict from nonlinear relationships, XOR
- Intuitively: learns one “template” per class, then measures correlation between template and input image
 - Drawback: Cannot handle multiple modes of data, intra-class variation
- Variations
 - Linear regression: $y = Wx + b$
 - Logistic regression: $y = \sigma(Wx + b)$
 - Softmax regression: $z = Wx + b \implies y_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$

Finding a good W : via a **loss function** to quantify performance of W

- Loss function averaged across all samples in dataset
- Example loss functions
 - Linear regression: L2 distance $\frac{1}{2}(s_i - y_i)^2$
 - Logistic regression: cross-entropy loss $L = -\log\left(\frac{\exp(s_{y_i})}{\sum_j \exp(s_j)}\right)$
 - Log of predicted probability (via softmax) of true class
 - Binary cross-entropy/BCE: $BCE(p, y) = -(y \log(p) + (1 - y) \log(1 - p))$
 - Multiclass SVM: hinge loss $L = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$
 - Sum of predicted score - true score, max with 0 (wants to make true class' score higher than all other classes)

Optimization: Finding the set of weights minimizing loss

- Via **gradient descent** (negative gradient - direction of steepest descent)

- Computing gradients: **numeric gradient** vs **analytic gradient**
 - **Numeric gradient** - approximate & slow convergence, but easier to implement
 - **Analytic gradient** - exact & fast, but more error-prone
 - Used in practice; numeric gradient used as a check (gradient check)
- **Gradient descent**: at each step, move in the direction of negative gradient
 - Hyperparameters: weight initialization method, # steps, learning rate
 - Challenges: gradient may become zero or vanish, may become stuck in local mins
- **Interpretation**: loss over dataset computes the expected value of loss over real-world distribution of values (x, y)

Issue: *Batch gradient descent* (computing gradient over full batch) is expensive for large batches

- Can use **stochastic gradient descent/SGD** - GD on minibatches (batch size as hyperparameter)
- Normal SGD results in noisy gradient; can use **momentum** (add a weighted running mean of gradients to SGD update) for less noisy optimization
 - Momentum: $v_t = \nabla f(x_t) \mapsto v_t = \rho v_t + \nabla f(x_t)$

AdaGrad: scale gradient element-wise based on historic sum of squares

- $s_t = s_{t-1} + g_t^2 \implies w_t = w_{t-1} - \frac{\eta}{\sqrt{s_t + \epsilon}} g(t)$
- Acts as a form of per-parameter/adaptive learning rate
- Lowers magnitude of steps along “steep” directions; boosts magnitude of steps along “flat” directions
- **RMSProp/weighted AdaGrad**: adds a decay rate for historic gradient term
 - $s_t = \gamma s_t + (1 - \gamma) g_t^2$
- **Adam**: “almost” RMSProp + momentum
 - Update step is (almost) momentum SGD update with RMSProp magnitude scaling
 - Bias correction - accounts for first/second moment estimates starting at 0
 - Divides momentum, RMSProp terms by $1 - \beta_1^t, 1 - \beta_2^t$ respectively

Adam:

```

input :  $\gamma$  ( $\text{lr}$ ),  $\beta_1, \beta_2$  (betas),  $\theta_0$  (params),  $f(\theta)$  (objective)
initialize :  $m_0 \leftarrow 0$  (first moment),  $v_0 \leftarrow 0$  (second moment)

```

```

for  $t = 1$  to  $\dots$  do
   $g_t \leftarrow \nabla_{\theta} f_t(\theta_{t-1})$ 
   $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$ 
   $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$ 
   $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ 
   $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ 
   $\theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ 

```

```

return  $\theta_t$ 

```

Overfitting occurs when model performs well on training data, but poorly on unseen data

→ **Regularization** incorporates an additional term $\lambda R(W)$ in loss function to incentivize less complex learned models; helps to combat overfitting

- Decreases performance on training set in exchange for better performance on validation, test sets
- Simple approach: L1/L2 regularization on magnitude of weights
 - More complex: dropout, batchnorm, cutout, mixup, etc.
 - *Interpretation*: encodes some form of “preference” regarding model weights
 - Ex: L2 prefers more “spread out” weights
- Note: L2 regularization (encodes penalty in loss itself) vs weight decay (adds additional $-\lambda w$ term in update step directly)
 - Equivalent for SGD + variants, but different for adaptive methods (e.g. AdaGrad) - weight decay is not included in adaptive sums, but L2 is

2nd-order optimization: rather than just the gradient, use gradient + Hessian to make quadratic approximation of loss landscape (look at 2nd-order Taylor expansion, use Newton’s method)

- Issue: Inverting Hessian (Newton’s method) is expensive, $O(N^3)$
- Can use Quasi-Newton methods: approximate inverse Hessian via rank-1 updates $O(N^2)$
 - L-BFGS - doesn’t store full inverse Hessian (low memory use), but only works well for full batch setting; does not transfer well to minibatch
- In practice: 1st-order in most cases; L-BFGS when doing full batch updates

Machine Learning (Misc.)

- **Hyperparameters:** choices made before learning regarding parameters (rather than being learned from data)
 - Evaluate via split train/validation/test datasets
 - k-fold cross validation for small datasets
- **Curse of dimensionality:** Number of points needed for uniform coverage of space increases exponentially with input dimension
- **Normalization:** can normalize datasets relative to known statistics (mean + stdev, e.g.) for easier learning & better transfer across datasets

Neural Networks

Issue: Linear classifiers can only learn linear decision boundaries

- Can use feature transforms - linear boundaries in transformed space correspond to nonlinear boundaries in original space
 - *Issue:* Requires knowing (fixed) feature transforms beforehand
 - Ex: color histogram/histogram of oriented gradients (HoG) as image features
- Image features via bag of words/BoW approach - extract random patches from image & cluster to form “codebook of visual words”
 - Use codebook as image encoding, a new ML model (e.g. SVM) for classification

Neural networks incorporate a nonlinear **activation function** between linear layers

- **Multi-layer perceptron/MLP** - use multiple fully-connected layers (rather than regression/single-layer)
 - Later layers use features from previous layer’s activation
 - Weakly inspired by brain structure
 - Universal approximator (similar to k-NN)
- Ex. (Activation functions)
 - **ReLU:** $f(x) = \max(0, x)$ / **Leaky ReLU:** $f(x) = \max(0.2x, x)$
 - **Sigmoid:** $\sigma(x) = 1/(1 + e^{-x})$
 - **Tanh:** $\tanh(x) = (e^{2x} - 1)/(e^{2x} + 1)$
 - **Softplus:** $\log(1 + \exp(x))$

Convex functions: intuitively, look like a “multidimensional bowl”

- Generally easy to optimize, can derive convergence guarantees
 - In optimization, prefer to minimize convex functions if possible
- Linear classifier loss functions are convex; but neural networks are generally non-convex (few/no convergence guarantees, but decent empirical behavior)

Backpropagation

When training neural networks, need to compute gradients

- Can construct *computational graphs* - indicating operations from inputs to outputs
- **Backpropagation**: can use the Chain Rule to derive gradients of the loss function with respect to every weight in network
 - From weights at end of network, “backpropagate” derivatives to further-back weights (from last layer to first layer)
 - Chain rule: Downstream gradient $\frac{\partial f}{\partial y} = \frac{\partial q}{\partial y} \cdot \frac{\partial f}{\partial q}$ (local * upstream gradients)
 - First perform forward pass (computing $f(x, w)$), then use backward pass to compute gradients for every weight
- Patterns in gradient flow:
 - Add gates distribute the upstream gradient to both elements of input
 - Copy gate adds upstream gradients to find singular downstream gradient
 - Multiplier gate multiplies upstream gradient with multiplicands
 - Max gate gives downstream gradient equal to upstream (for taken branch), 0 for branches not taken
- Pytorch *autograd* - uses forward, backward methods to compute gradients automatically

Backpropagation with vectors: rather than local gradients (i.e. vectors), multiply by local Jacobian matrices (2D) instead

- Jacobians indicate how much each element of inputs influence output
- Jacobians are sparse (all off-diagonal entries 0), large; therefore, can compute matrix product implicitly (via normal multiplication operators, without constructing full matrix)
- For backprop with matrices, use multi-dimensional Jacobian

Convolutional Neural Networks

Image classification: Given an image and a set of K possible classes, output the class that matches that image

- A fundamental CV task
- Challenges: viewpoint & intraclass variations, interclass similarities, occlusion, domain changes, etc.

Image classifiers

- *k*-Nearest Neighbors/*k*-NN: Find k nearest points to input in the memorized dataset
 - O(1) training, O(N) testing
 - Image classification via pixel distance (questionable metric)
 - Alt: k-NN on ConvNet features (works well)
 - Is a universal approximator (can represent [almost] any function)
- Regular neural networks (MLPs)
- Convolutional neural networks (CNNs)

Convolution & Normalization Layers

Issue: Regular neural networks (i.e. MLPs) don't explicitly consider the spatial structure of images; act only via local matrix/vector products + activation function

- Within an MLP, have no interaction between adjacent input elements within a layer

Solution: can define new spatial operations tailored to image format: **convolution & pooling layers**

Convolution layers convolve input image with a filter/kernel

- Filter contains same depth/# channels as input image; "slide" over image spatially (take dot products) to produce *activation map*
 - Kernel dimension corresponds to number of dimensions of input (e.g. 2D input gives 2D kernel; 3D input gives 3D kernel)
 - Can use multiple filters; each produces a single channel in output
 - Also use a bias vector (one scalar per filter, same size as activation map)
 - Shapes: $(N, C_{in}, H, W) \rightarrow (N, C_{out}, H', W')$

- Convolution as cross-correlation: dot product performs a matching between filter, scanned elements (higher -> better match)

Can stack multiple convolutional layers (with activations in between)

- Multiple convolution layers, stacked, correspond to a single larger convolution
 - Each convolution layer is a linear classifier
- Add **padding** around input (consisting of zeros) to preserve size of feature map with each convolution layer
 - Can set $P = (K - 1)/2$ [K the kernel size] to preserve input shape
- Further-back convolution layers correspond to convolutions of/depend on larger portions of the region (have larger receptive fields)
 - Each convolution adds $K - 1$ to size of **receptive field**
 - Receptive field size: $1 + L \cdot (K - 1)$
 - Initial layers learn local features (e.g. local image templates, edges); higher layers learn more complex features
- **Issue:** for large images, need many layers for receptive fields to “see” whole image
 - Can downsample inside network to reduce # layers needed
 - **Strided convolutions** - convolutions take larger steps between dot products to produce a smaller activation map
 - For later layers in network: spatial size decreases (via pooling), but number of channels increases (preserving total volume)
- Output size: $H' = (H - K + 2P)/S + 1$ (and similar for W')

Pooling layers: alternative way to downsample feature map

- **Max pooling** - takes maximum of values within receptive field as output value
 - Results in invariance to small spatial shifts
- Hyperparameters: kernel size, stride, pooling function
 - Output size: $H' = (H - K)/S + 1$

Convolutional Neural Networks (CNNs)

Classic architecture (LeNet/AlexNet):

1. [Conv, ReLU, Pool] N times

2. Flatten
3. [FC, ReLU] N times
4. FC to produce output

Issue: Deep networks are difficult to train, susceptible to “*internal covariate shift*”

Solution: Use **batch normalization** - normalize layer outputs to make them have zero mean, unit variance across a batch

- Formula: $\hat{x} = \frac{x - E[x]}{\sqrt{Var[x]}}$ (differentiable, for backprop)
 - Each dimension of input x is normalized separately
 - Use (running) average μ_j, σ_j^2 of mean/variance values observed in training during testing
- Can learn scale, shift parameters γ, β [dim D] to keep more information than pure zero mean/unit variance
 - Final output: $y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$
- Batch norm for convolutional networks - *spatial batchnorm*
 - Perform batch-norm across each channel (for (N, C, H, W) shape, perform batchnorm across slices (N, H, W))
- Makes deep networks much easier to train:
 - Allows higher learning rates, faster convergence
 - Makes networks more robust to initialization
 - Acts as a form of regularization during training
 - During testing (parameters fixed), is a linear operator; can incorporate into convolutional layer directly
- *Issues:* not well-understood theoretically, can behave weirdly during training/testing (causes bugs)

Computing parameters

- Number of params - sum of total number of weights in network + number of biases
 - Single convolutional layer: $C_{out} \cdot C_{in} \cdot K \cdot K + C_{out}$
- 4 bytes per element (per float) gives memory
- Floating point operations/FLOPs: number of output elements * ops per output element

- Single convolutional layer: $(C_{out} \times H' \times W') \cdot (C_{in} \times K \times K)$

Types of normalization layers

1. Batch normalization (above)
2. *Layer (1D) normalization*: for each input in batch, compute mean/variance across entire dimension of input (channel + height, width, etc.)
3. *Instance (2D) normalization*: for each input in batch, compute mean/variance (for each channel, separately) across entire dimension of input

Inductive bias - encoding some hypothesis about network function into network architecture

Modern CNN Architectures

AlexNet (2012) - [Conv, ReLU, Pool] into [FC, ReLU]

- First major instance of deep CNNs for classification, achieved 1st place on ImageNet
 - From LeNet: deeper & larger, ReLU instead of sigmoid activation, larger dataset + more training cycles
 - Architecture (i.e. layer kernel sizes, e.g.) via trial and error
- Notable characteristics:
 - Most memory usage in early convolutional layers; FLOPs in convolutional layers
 - Nearly all parameters in fully-connected layers
- Succeeded by similar, deeper networks (**ZFNet**, **VGG**)
 - **ZFNet**: changed kernel stride sizes
 - **VGG**: more regular design: all conv are 3x3/pad 1, max pool 2x2 stride 2; double # channels after each pool
 - 2 3x3 convolutions have same receptive field as 5x5, but fewer parameters & faster to compute
 - Significantly larger than AlexNet

GoogLeNet - introduced improvements on AlexNet

- Aggressive downsampling at input via *stem network* to decrease computation, memory
- *Inception module* - local unit with parallel branches (multiple parallel convolutional layers with different dimensions, followed by concatenation); repeated throughout network
- **Global average pooling (GAP)** followed by linear layer (rather than FC layers) at end; requires significantly less parameters
- Hack: auxiliary classifiers (intermediate “classifiers” computing loss) in middle stages of network
 - Since propagating loss across entire network depth is not clean process (before batchnorm)

ResNet

Issue: With batchnorm, can train deeper and deeper networks; however, CNN (i.e. AlexNet & similar) performance actually decreases with deeper models (rather than shallow models)

- Initial guess was that model was overfitting; however, training error is also worse on deeper models (not just test) -> deeper models are underfitting
- Hypothesis: deeper models harder to optimize, can't easily learn identity functions to emulate shallow models
 - Deeper models should be able to learn shallow model + successive layers of identity to match shallower models' training error

Residual networks add an extra “additive shortcut” between convolutional blocks to make learning identity functions easier

- **Residual blocks:** $x \mapsto f(x)$ becomes $x \mapsto f(x) + x$
 - Each residual block - first block halves resolution, doubles # channel

ResNet - stack of residual blocks (similar to VGG)

- Like GoogleNet - stem network at start for downsampling, global average pooling to replace FC layers at end
- **Bottleneck block:** replaces two 3x3 convs (within each residual block) with 1x1 -> 3x3 -> 1x1 convolutions
 - More layers for less computational cost
 - ResNet-50 - replaces ResNet-34 basic blocks with bottleneck blocks
- ResNets are able to train very deep networks, perform better than shallow ones
 - Still widely used today

Complexity of convolutional models

- VGG - highest memory + most operations
- GoogLeNet - very efficient (low # ops)
- AlexNet - low # operations, large # parameters
- ResNet - moderately efficient (operations + parameters), higher accuracy
 - Can train very deep networks (to the point of diminishing returns)
 - Good standard choice of architecture
- Later: vision transformer (ViT) matches ResNets, outperform with more data

Improving residual networks

- Block design
 - Pre-activation: ReLU inside residual (rather than after) to ensure block can learn true identity
- ResNeXt - rather than single path of convolutions within bottleneck block, compute G-many paths in parallel, add together with residual at end
 - Grouped convolution - rather than all convolutional kernels touching all channels, have parallel convolution layers each working on a subset of channels (e.g. $\frac{\# \text{channels}}{N}$)
 - Each parallel layer produces $\frac{C_{\text{out}}}{N}$ output channels
 - Allows for parallelization, distributing RAM cost
 - ResNeXt - add groups to improve performance, maintain computational complexity
- Squeeze-and-Excitation/SENet - adds “squeeze and excite” branch (global pooling + $2 \times \text{FC}$ + sigmoid) within each residual block; multiplies with residual output
 - Adds form of global context to each residual block

Other forms of convolutional networks

- *Densely connected neural networks*: introduce dense blocks, where each layer is connected to every other layer (essentially: concatenate outputs from all past layers to current layer's input at every step)
 - Alleviates vanishing gradient by strengthening propagation + encourages feature reuse across layers
- *MobileNets* - tiny networks (memory-wise)
 - Replaces standard convolution block with “depthwise convolution” (grouped convolution, $\# \text{ groups} = \# \text{ channels}$) block + pointwise 1×1 conv block
 - Reduces number of parameters dramatically (divides by C)
 - Used in mobile devices; related: ShuffleNet

Neural architecture search - automated process for designing NN architectures

- Controller network outputs network architecture, samples & trains child networks from controller; after each batch, perform policy gradient step on controller & repeat
- Outputs good architectures after a long time, but very expensive to perform
 - Can use to find efficient CNN architectures

Training Neural Networks

Initialization

Notable activation functions:

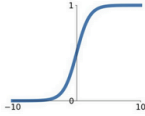
1. **Tanh** - squashes numbers to $[-1, 1]$
 - a. Zero-centered (good), but kills gradients when saturated (bad)
2. **ReLU**
 - a. Advantages: does not saturate in + region, very efficient, converges fast in practice
 - b. Disadvantages: no zero-centered output, gradient 0 for all $x < 0$ (no updates)
3. **Leaky ReLU**
 - a. Advantages of ReLU + no vanishing gradients
4. **Exponential Linear Unit/ELU**: $f(x) = x$ for $x > 0$, $f(x) = \alpha(e^x - 1)$ for $x \leq 0$
 - a. Benefits of ReLU + closer to zero mean + negative saturation regime compared to leaky ReLU (some robustness to noise)
 - b. More expensive to compute (requires exp)
5. **Scaled Exponential Linear Unit/SELU**
 - a. $selu(x) = \lambda x$ if $x > 0$, $selu(x) = \lambda \alpha(e^x - 1)$ if $x \leq 0$
 - b. Scaled version of ELU, better for deep networks; is “self-normalizing” (can train deep SELU networks without batchnorm)
6. **Gaussian Error Linear Unit/GELU**:
$$X \sim N(0, 1) \implies gelu(x) = xP(X \leq x) = \frac{x}{2}(1 + erf(x/\sqrt{2})) \approx x\sigma(1.702x)$$
 - a. $x\sigma(1.702x)$
 - b. Multiplies input by 0 or 1 at random; large values more likely to be multiplied by 1, small values by 0 (data-dependent dropout)
 - c. Common in transformers

ReLU used in most cases (Leaky ReLU/ELU/SELU in limited cases)

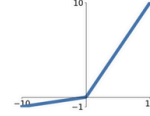
- Sigmoid, tanh not generally used except to squash the output

Sigmoid

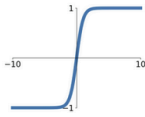
$$\sigma(x) = \frac{1}{1+e^{-x}}$$

**Leaky ReLU**

$$\max(0.1x, x)$$

**tanh**

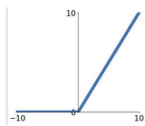
$$\tanh(x)$$

**Maxout**

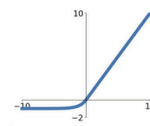
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ReLU

$$\max(0, x)$$

**ELU**

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

**Weight Initialization:** How to initialize weights?

- Initialize all 0 causes all gradients to be the same; no “symmetry breaking”
- Initialize with small random numbers (Gaussian with zero mean, 0.01 stdev) works okay for small networks, but has problems for deeper networks
 - All activations tend to zero for deeper network layers
- Similar, but with 0.05 stdev -> all gradients saturate (local gradients zero, no learning)
- “Xavier initialization” - stdev = $1/\sqrt{\text{Din}}$ - scales activations nicely for all layers
 - Conv layers: Din is $K^2 \cdot C_{in}$
 - Derivation: Sets value such that variance of output = variance of input (assuming x, w are i.i.d. zero-mean)
- ReLU initialization: Xavier assumes zero-centered activation function -> doesn't work, gradients vanish
 - *Kaiming initialization:* stdev = $\sqrt{2/\text{Din}}$ works well
- For residual networks, initializing with Kaiming method causes variance to grow with each block (due to residual connection)
 - *Solution:* first conv with Kaiming, second conv to zero in each residual block

Regularization

- Commonly used: L2 regularization, L1 regularization, Elastic net (L1 + L2)
- **Dropout:** in each forward pass, randomly set some neurons to 0
 - Probability of dropping as hyperparameter; commonly 0.5

- Forces network to have redundant representation; prevents co-adaptation of features (i.e. each neuron in a layer encoding an entirely separate feature)
 - Alt: dropout trains a large ensemble of models that share parameters
- Used on FC layers for early networks, e.g. AlexNet/VGG (where most parameters located); GoogLeNet, ResNet, etc. use global average pooling (no dropout needed)
- **Data augmentation:** when training for image classification, can perform various transformations to image (simulates training on a larger dataset)
 - Ex: horizontal flip, random crops/scales
 - More complex: color jitter, shearing, lens distortions, etc.

Intuitively, augmentation adds some randomness during training

- More approaches:
 - *Stochastic depth*: skip some residual blocks in ResNet during training
 - Use whole network during testing
 - *Cutout*: set random regions of images to 0 during training
 - Works well for small datasets; less common for larger datasets
 - *Mixup* - train on random blends of images
 - Ex: 40% dog, 60% cat (pixel-wise) has target label 0.4 dog, 0.6 cat
 - Scale blend probability from a beta distribution $\text{Beta}(a,b)$ to keep blend weights close to 0, 1

General lessons for training NNs:

1. Batch normalization/data augmentation generally good ideas
2. Dropout for large FC layers

Training

Learning rate schedules: rather than a fixed learning rate, can instead vary learning rates over time

- **Step** (most common): reduce learning rate at a few fixed points
 - Ex: ResNet multiplies LR by 0.1 after epochs 30, 60, 90
- **Cosine:** $\alpha_t = \frac{1}{2}\alpha_0 \left(1 + \cos\left(\frac{t\pi}{T}\right)\right)$
- **Linear:** $\alpha_t = \alpha_0 \left(1 - \frac{t}{T}\right)$
- **Inverse sqrt:** $\alpha_t = \alpha_0 / \sqrt{t}$

How long to train?: Want to avoid overfitting

- **Early stopping** - stop training model when accuracy on validation set decreases
 - Alt: keep training, but store model snapshot that worked best on validation
- Alt approach: first stop training when accuracy on validation set decreases, and record that iteration #; then train train + val dataset together, stopping at previous best iteration

Choosing Hyperparameters

Can model hyperparameter choice as Bayesian optimization problem

Grid search: can choose several values for each hyperparameter (e.g. learning rate + weight decay), evaluate all possible choices on the hyperparameter grid

- Often space choices log-linearly
- Alt: via random search (log-uniform on a certain interval, run many different trials)
 - In case where one parameter is known to be more important than the other: random search samples more values of that parameter than grid
- Facebook: one single learning rate & weight decay works well across many different models/architectures (via large-scale empirical random search)

Choosing hyperparameters (in general)

1. First, check initial loss (turn off weight decay, sanity check loss at initialization)
2. Next, overfit a small sample (try to train to 100% accuracy on a small sample of training data, e.g. 5-10 minibatches)
 - a. Use to tune architecture, LR, weight initialization (no regularization yet) until loss zeroes out
3. Use architecture from previous step with all training data + small weight decay, use to tune learning rate
 - a. Want learning rate that makes loss drop significantly within ~100 iterations
4. Use coarse grid, train for 1-5 epochs to test learning rate, weight decay
5. Refine grid, train longer without learning rate decay
6. Look at learning curves to qualitatively assess training progress/challenges & repeat

After Training

Model ensembles: can train multiple independent models and average results at test time for a slight boost in performance

- Take average of predicted probabilities & argmax

Transfer Learning

Rather than using a lot of data to train/use CNNs, use a pretrained CNN on a standard dataset (e.g. ImageNet) and use extracted features (before FC layers) on a new dataset

- Remove FC layers at end of CNN & replace with a new set of FCs (using same conv layers + weights as before); then, train CNN for new task
 - Works well for training on new datasets without having to retrain feature extractor
 - Can lower learning rate (e.g. 1/10 of LR used in original training), freeze lower layers during fine-tuning process
- Amount of additional layers + finetuning depends on size of new dataset, similarity to ImageNet
 - For very different datasets + small dataset, may need to add more layers or cut off & retrain some earlier layers
- Allows for transferring improvements in CNN architectures from image classification, e.g., to downstream tasks by replacing CNN feature extractor with a newer one
 - Can use as feature extractor for object detection, e.g.
- Reduces training time needed

Another approach - rather than training the CNN feature extractor on a large labeled dataset (hard to find), can instead train on large unlabeled datasets (unsupervised representation learning)

- Weakly supervised learning

Visualizing & Understanding Neural Networks

3 perspectives:

- Understanding network as a whole
- Looking at feature spaces
- Looking at individual units

Can visualize convolutional filters in CNN

- Lower layers correspond to local features of image
- Higher layers - not interesting

Looking at features - collect feature vectors from running network on many images, then use k-NN to compare features in feature space

- Can use dimensionality reduction (e.g. PCA) to visualize

Maximally activating patches - pick a layer & channel, then run many images through the network and find the image patches that have the highest value for that channel

- Use guided backprop to visualize

Annotating interpretation of images

- Within classified image, cut out the region (during annotation) corresponding to a label
- Can dissect networks to find “interpretable units” corresponding to each label; use IoU to evaluate (semantic segmentation)

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

Saliency - want to determine which pixels most affect the output

- **Saliency via occlusion:** Mask part of image before feeding to CNN & determine how probabilities change
- **Saliency via backprop:** Compute probabilities (forward pass) and compute gradient of (unnormalized) class score with respect to each pixel (absolute value)
- Saliency maps act as form of segmentation without supervision

- Not a perfect interpretability metric

Class activation mapping (CAM)

- Rather than summing all entries (across the image) in final FC layer to produce a given class's score, simply output matrix of final FC values as that class's CAM
- Can use as a form of weakly-supervised localization/object detection, interpretability
 - Only applies to last conv layer, but recent CNNs use global average pooling anyway

Gradient-weighted class activation mapping (Grad-CAM)

- Generalization of CAM:
 - Pick any layer's activations, compute gradient of class score with respect to activations
 - Use GAP on gradients to get weights, use ReLU to compute activation map
- Can also utilize for other kinds of vision models (e.g. image captioning)

Visualizing CNN features: *gradient ascent*

- Computes the synthetic image that maximally activates a neuron
 - Starting with zero image, keep forward passing & stepping input image in direction of positive gradient (of neuron value w.r.t. image pixels)
 - L2 regularization on generated image pixels; also periodically Gaussian blur + clip pixels with small values to 0, small gradients to 0
 - Can use to visualize intermediate features at each layer
 - "Multifaceted visualization" (using more careful regularization, enter bias) for even nicer results
 - Can perform gradient ascent by training a deep generator network (prior) for the synthetic image, followed by the CNN
 - Can backpropagate through the CNN and generator network
- Adversarial usage: given an arbitrary image, start with an arbitrary category and modify image (via gradient ascent) to maximize class score until network is fooled

Feature inversion - given a CNN feature vector for an image, find a new image with a similar feature vector that "looks natural" (via some regularization)

DeepDream - try to amplify neuron activations at some layer in the network

- Given an image and layer: perform forward pass, then set gradient of chosen layer equal to its activation & backprop
- Trippy output

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

- Via nearest neighbors: generate pixels one at a time in scanline order; form neighborhood of already-generated pixels and copy NN from input
- Via neural networks - reshape features from $C \times H \times W$ to $C \times HW$ and compute Gram matrix $G = FF^T$
 - Neural texture synthesis: use a pretrained CNN & run input texture forward through CNN, recording activations on every layer
 - At each layer, compute Gram matrix
 - Initialize generated image from random noise, pass through CNN, compute Gram matrices
 - Compute weighted sum of L2 distance between Gram matrices, backprop to get gradient on image, and make gradient step (& repeat)
- Higher layers recover larger features from input texture
- Can use for style transfer - texture transfer + Gram reconstruction
 - Matches features from content image / Gram matrices from style image
 - Adjust weight-to-style loss coefficients
 - Can mix style from multiple images via weighted average of Gram matrices

Style transfer requires many forward/backward passes through VGG (slow)

- Solution: train another neural network to perform style transfer
 - Fast neural style transfer - train a feedforward network for each style, compute same losses as before -> stylize images via single forward pass (after training)
 - Uses instance normalization
- Can train one network for multiple styles via conditional instance normalization (learning separate scale, shift parameters per style)

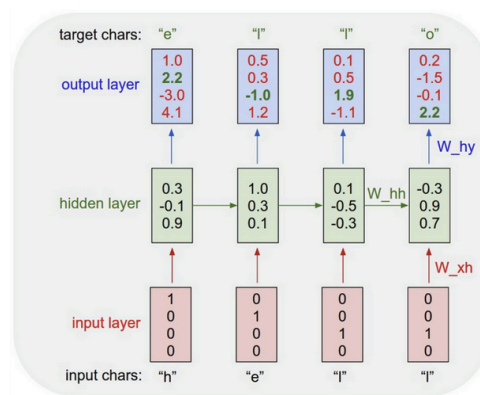
RNNs & Attention

Rather than “one-to-one”/feedforward models (seen previously), look at *sequence models* (one to many/many-to-one/many-to-many)

- Ex: for image captioning (image -> sequence of words), video classification (sequence-to-label), machine translation (sequence-to-sequence)
- Per-frame video classification

Recurrent neural networks (RNNs) - store an “internal state”, update as sequence is processed

- New state is function of old state & input vector at some time step $h_t = f_W(h_{t-1}, x_t)$
- Same function (i.e. model) f_W , set of parameters W at every time step



Vanilla RNN - state consists of a single hidden vector h_t

- $h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t + b_h)$, $y_t = W_{hy}h_t + b_y$
- Ex (language modeling): Given characters $1, 2, t-1$, model predicts character t
 - Character at time t is function of hidden state + preceding character; train model to correctly predict from existing training sequences (strings)
 - At test time: generate new text; sample characters one-by-one and feed back to model
 - At each step: input to model is output from previous sample step
 - Start with an artificial character <START>; cut off sequence when model samples <END> character
- Ex (image captioning): use feature vector from a CNN as initial hidden state for an RNN

Attention

RNNs for sequence-to-sequence prediction

1. Initially: continuously update hidden state on all characters of input sequence
2. From final hidden state, predict initial decoder state s_0 and context vector c (e.g. $c = h_T$)
3. Decoder uses context vector c , START character y_0 (plus initial state s_0) to determine decoder state s_1
4. Afterward: Keep updating decoder state $s_t = g_U(y_{t-1}, s_{t-1}, c)$ and sampling another character y_t until STOP character

Issue: input sequence bottlenecked through a fixed-size vector; may not be enough for fixed sequences

Solution: Use a new context vector at each step of decoder

At each step:

1. Use decoder state s_t to compute alignment scores $e_{t,i} = f_{att}(s_{t-1}, h_i)$ (using MLP f_{att})
 - a. Initial decoder state: predicted from final hidden state h_n
2. Use softmax on alignment scores to get attention weights $a_{t,i}$
3. Compute context vector as linear combination of hidden states $c_t = \sum_i a_{t,i} h_i$
 - a. h_1, \dots, h_n one hidden state per input element
4. Use context vector as input to decoder: $s_t = g_U(y_{t-1}, s_{t-1}, c_t)$
 - a. Sample y_t from decoder

Intuition: context vector “attends to” (assigns higher weights $a_{t,i}$ to) the relevant part of input sequence [determined by MLP]

- Input sequence no longer bottlenecked through a single context vector; at each decoder timestep, context vector “looks at” different parts of input sequence
- Not limited to language - can use for any set of input hidden vectors h_j
 - Decoder doesn't use ordered nature of h_j 's anywhere; can be unordered

Image captioning with RNNs and attention

- Use CNN to compute grid of features $h_{i,j}$ for image

- At each decoder timestep: compute alignment score for each feature $h_{i,j}$ and use to find new context vector

Can use attention for interpretability

- Machine translation - high attention weights correspond to relevant parts of input sentence at the current timestep/stage in translation
- Image captioning - attention weights for each word in caption correspond to the relevant part of image
- *Intuition*: works like human eye attention, picks region of focus

Types of Attention

- **Attention layer**: uses scaled dot product to compute similarities $e = Q \cdot X / \sqrt{D_Q}$ between query vector(s) Q , input vectors X
 - Machine translation: singular query vector is decoder state; input vectors are hidden states
 - Can generalize to multiple query vectors (dot product to matrix product)
 - Scaled dot product - multiplies dot product by sqrt of query vector dimension
- **Attention layer (key-value)**
 - Add new **key matrix** W_K , **value matrix** W_V
 - Compute keys by $K = XW_K$, values by $V = XW_V$
 - Take similarities by product between query vectors Q and keys K ; output vectors by $Y = AV$ (A attention weights)
- **Self-attention**: uses query matrix W_Q to compute query vectors by $Q = XW_Q$, in addition to key & value vectors
 - One query per input vector
 - Output is permutation equivariant (permuting input gives permuted version of same output)
 - Self-attention doesn't consider order of processed vectors
 - Solution: can concatenate input with a positional encoding [vector] E to make processing "position-aware"
 - Encoding may be learned or via fixed function
- Masked self-attention (for language modeling - predicting next word)

- Zero out all similarities with keys “further ahead” of current query vector
 - Prevents vectors from “looking ahead” in the sequence
- *Multihead self-attention*: use H independent self-attention heads in parallel
 - Copy input for each independent head; concatenate output vectors at end
 - Hyperparameters: query dimension D_Q , number of heads H

Self-Attention Layer

One query per input vector

Inputs:

Input vectors: X (Shape: $N_X \times D_X$)

Key matrix: W_K (Shape: $D_X \times D_Q$)

Value matrix: W_V (Shape: $D_X \times D_V$)

Query matrix: W_Q (Shape: $D_X \times D_Q$)

Computation:

Query vectors: $Q = XW_Q$

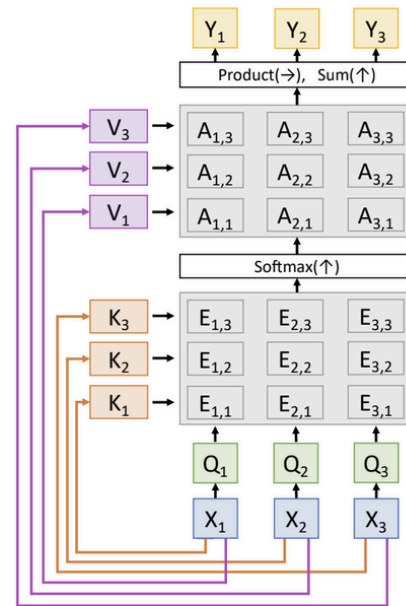
Key vectors: $K = XW_K$ (Shape: $N_X \times D_Q$)

Value Vectors: $V = XW_V$ (Shape: $N_X \times D_V$)

Similarities: $E = QK^T / \sqrt{D_Q}$ (Shape: $N_X \times N_X$) $E_{i,j} = (Q_i \cdot K_j) / \sqrt{D_Q}$

Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ (Shape: $N_X \times N_X$)

Output vectors: $Y = AV$ (Shape: $N_X \times D_V$) $Y_i = \sum_j A_{i,j} V_j$



CNN with self-attention

- Query, key, value matrices given by 1×1 conv with dim $C' \times H \times W$
- Use standard self-attention module plus residual connection

Three ways to process sequences

- RNN: works on ordered sequences
 - Good at long sequences (h_T “sees” whole sequence after one RNN layer), but not parallelizable (requires on sequentially finding hidden states)
- 1D convolution: works on multidimensional grids
 - Bad at long sequences (need to stack many layers for receptive field to include whole sequence), but highly parallel
- Self-attention: works on sets of vectors
 - Good at long sequences and highly parallel, but very memory-intensive

Transformers

Transformer block:

1. Self-attention (with residual connection)
2. Layer normalization
3. MLP independently on each vector (with residual connection)
4. Layer normalization & output

Transformers - sequences of transformer blocks

- Highly scalable & parallelizable
 - Only interaction between vectors happens within self-attention layers
- Multi-head attention: module for attention, runs through attention mechanisms several times in parallel & concatenates (independent) attention outputs
 - Uses a linear layer on final output to transform into expected dimension
- Effect similar to AlexNet for NLP
 - Use pretrained transformers for NLP tasks

Self-attention vs cross-attention

- Cross-attention: rather than computing queries/keys/values all from input, obtain queries from decoder and keys/values from encoder
 - Translation: decoder contains information about target language statistics; encoder contains information about source language

Attention/transformers for vision

1. Add attention to existing CNNs/CNN architectures (add between existing ResNet blocks)
 - a. Still a CNN; want to replace convolution entirely
2. Replace convolution with “local attention”
 - a. Attention takes center of receptive field as query, surrounding elements to find keys/values; output computed via attention
 - b. Issue: hard to implement, only marginally better than ResNets
3. Standard transformer on pixels

- a. Issue: very memory-intensive, $R \times R$ image gives R^4 elements per att matrix
- 4. **Vision Transformer (ViT)**: standard transformer on patches
 - a. Divide image into smaller patches (flattened)
 - i. Concatenate each patch with a learned position embedding
 - b. Pass image patches as input to a standard transformer
 - c. Add special extra input (classification token vector, learned, same dim as image patch); take output vector corresponding to classification token as vector of class scores
 - i. No convolution layers needed (besides transformer MLPs)

ViT vs ResNet

- ViT performs worse on smaller datasets, but performs/scales better for larger ViTs + larger datasets
 - ViT advantage - better scalability (fairly efficient transformer memory-wise, faster to train)
- Most CNNs (e.g. ResNet): decrease resolution, increase channels for deeper layers; hierarchical structure
 - ViT: all block share same resolution, # channels; isotropic structure

Improving ViT

- Regularization
 - Weight decay, stochastic depth; dropout in transformer MLP layers
- Data augmentation: MixUp, RandAugment
- Distillation:
 - Train a teacher model to classify images from ground-truth labels
 - Train a student model to match predictions from the teacher (sometimes: also ground truth labels)
 - Easier than training student from scratch
 - Can also train student on unlabeled data for semi-supervised learning
 - Can train teacher CNN -> student ViT

Hierarchical ViT: Swin transformer

- First divide image into very small patches (e.g. 4x4), then merge (halve patch dimension) after every stage between transformer blocks
 - Merging: concatenate groups of 2x2 and linear project to half the channels
 - Results in hierarchical structure (similar to CNNs)
 - Other hierarchical transformers: MViT, Swin-V2, Improved MViT

Issue: matrices are big for earlier layers

Solution: don't use full attention, instead use attention over patches

- **Window attention** - rather than allowing each token to attend all other tokens, divide attention matrix into smaller $M \times M$ windows and only compute attention within each window
 - Linear in size for fixed M
 - Swin transformer - instead of positional embeddings (like ViT), encodes relative position between patches when computing attention
 - Adds bias term to similarity scores (before softmax)

Issue: no communication across windows; might lose information

Solution: alternate between normal windows and shifted windows (shifting all windows by some amount) in successive blocks

Faster & more accurate than previous models

- Can also use as backbone for downstream CV tasks (beyond classification)

Improving ViT

- ViT uses self-attention to mix across tokens; can try something simpler
 - MLP-Mixer: use MLP to mix across tokens (replacing self-attention)
 - All-MLP architecture
 - First applies an MLP across all N patches, then another MLP across all C channels

Object detection with transformers

- DETR (simple object detection pipeline): directly output set of boxes from transformer
 - Train using bipartite matching loss

- Uses transformer to encode image features, then another transformer to decode & generate output vectors; from output vectors, uses FFNs to generate prediction (no object, or class + box if object)
- *Diffusion Models with Transformers* (DiT): replaces latent diffusion U-Net backbone with transformer operating on latent patches

Computer Vision

Object Detection

CV tasks:

1. Image classification
2. Semantic segmentation
3. Object detection
4. Instance segmentation

Object detection: Given an image, output a set of detected objects (consists of a label + bounding box for each object)

- Bounding box represented via coordinates (x, y, w, h)
- Loss function: Take a weighted sum of label loss (softmax) & bounding box loss (L2) as final multitask loss
- *Challenges:* Multiple outputs, multiple types of output (label + bounding box), often works on higher resolution images (compared to classification)

Object detection models

- Take feature vector from vision backbone (pretrained ResNet, e.g.) and use separate FC layers for labels, box coordinates
 - Issue: images can have more than one object; need different numbers of outputs per image depending on image contents, # of objects
- Can detect multiple objects via sliding window: apply a CNN to many different crops of image to classify crop as object vs background
 - Bounding box is the window size
- *Issue:* too many possible windows to evaluate for large images
 - **Region proposals** - find a small set of boxes likely to cover all objects
 - Often based on heuristics (e.g. look for “blob-like” image regions)
 - Relatively fast (e.g. selective search)

Evaluating object detectors

- Can use **intersection over union (IoU/Jaccard index)** to compare prediction, ground-truth boxes
 - Formula: $(\text{Area of intersection}) / (\text{Area of union})$
 - Issue: object detectors often output overlapping detections
- Overlapping boxes (solution) - use **non-max suppression (NMS)**
 - Acts as form of post-processing: at each step, select the next highest-scoring box and eliminate all lower-scoring bounding boxes with a high IoU with current box
 - Issue: may eliminate good boxes if objects are highly overlapping (no easy solution)
- **Mean Average Precision (mAP)** - run object detector on all test images with NMS
 - For each category, compute average precision (AP) - area under Precision vs Recall curve
 - For each detection (highest to lowest score): if it matches some GT box with $\text{IoU} > 0.5$, mark as positive and eliminate GT box; otherwise, mark negative
 - Plot points on PR curve (precision vs recall graph)
 - Average precision: area under PR curve
 - AP = 1.0: hit all GT boxes with $\text{IoU} > 0.5$, and have no “false positive” detections ranked above any “true positives”
 - Mean Average Precision: average of AP for each category
 - COCO mAP: compute mAP for multiple IoU thresholds (0.5, 0.55, etc.) and take the average

Object Detection Models

Region-Based CNN (R-CNN): Takes regions of interest from proposal method, transform to standard CNN input size & forward regions through CNN for classifications

- Bounding box regression: from CNN features, predict a “transform” to correct RoI to produce bounding boxes
 - Can compare with ground-truth boxes
- Running R-CNN
 - For each proposal, resize to standard input size and run independently through CNN to predict class scores & bounding boxes
 - Use scores to select subset of region proposals to output

Issue: very slow, need to perform forward pass on many regions

→ *Solution:* Run CNN before warping

Fast R-CNN: Run entire image through a CNN, then run region proposal method on CNN-output image features

- Can use any vision backbone (e.g. ResNet)
- From proposed regions; crop & resize, run through a per-region CNN network to obtain category, box transform per region
 - Can have a heavy initial backbone and relatively lightweight per-region network (even just FCs) to save redundant computation
- Significantly faster to train & run than regular R-CNN
- Q: How to crop & resize features?
 - *Rol pool* - find proposal within input image, then project onto features and “snap” to grid cells
 - Can divide into 2x2 grid of roughly equal subregions and maxpool within each subregion - ensures that region features always have same size, even if the proposed regions have different sizes
 - Issue: “snapping” may cause misalignment; method also results in different-sized subregions (in some cases)
 - *Rol align* - find proposal within input image and project
 - Rather than “snapping”, can sample features at regularly-spaced points in each subregion using bilinear interpolation
 - At each point: look at distances to four neighboring grid cells; take feature value as weighted linear combination (no snapping needed)

Issue: most of the time running Fast R-CNN is spent finding region proposals (due to needing to run on CPU); want to find a way to learn instead

Faster R-CNN: incorporates learnable region proposal network to predict proposals from features

- From feature map, region proposal network predicts proposals
 - At each point in feature map: take a fixed-size “anchor box” around it and predict whether corresponding center point (anchor) contains an object
 - For positive boxes, also predict a box transform to convert from anchor box to object box
 - Issue: anchor box around a point may have wrong size/shape to include object

- Solution: Use K different anchor boxes (of different shapes/sizes) at each point
- Jointly train 4 losses: RPN classification & regression + object classification & regression
- Significantly faster to run than Faster R-CNN
- Is a two-stage object detector: (i) Backbone & RPN, run once per image; (ii) Classifier for proposed regions (run once per region)
 - Q: Do we really need the second stage?

Single-stage object detection (e.g. YOLO, SSD): instead of classifying anchors in RPN as object/not object, classify as one of C categories (or background)

- May also use category-specific regression: for each non-background category, also predict a bounding box transform
- Less accurate than two-stage methods, but much faster

Object detection

- Better backbones are slower, but perform better
 - Recent backbones: feature pyramid networks (multiscale backbone), ResNeXt
- Single-stage methods have improved
- Very big models work better
- Test-time augmentation can help boost performance
- ICCV '23: object detection as a diffusion process
 - Generate an image overlaid with a random set of boxes; diffusion model will progressively refine into a better box at inference time

Semantic Segmentation

Semantic segmentation: Label each pixel with a category label (without differentiating instances)

Initial idea: sliding window

- Within an image, take a certain crop around a central pixel and classify it
- Assign pixel category to be output class from classifier
- Issue: inefficient, doesn't reuse shared features between patches

Neural networks for semantic segmentation

First idea: vanilla CNN with constant kernel size

Issue: convolution expensive on large images + receptive field is linear in # of convolutional layers

New idea (**encoder-decoder**): structure CNN as two stages: **downsampling** -> **upsampling**

- **Downsampling** - use pooling, strided convolution to decrease resolution of features
- **Upsampling:** use **unpooling**, **transposed convolution** to increase feature resolution
 - **Unpooling** (e.g. 2x2 -> 4x4)
 - Naive approaches: simple (only fill in top-left corner), nearest neighbor
 - More complex: bilinear interpolation, bicubic interpolation, etc.
 - "Max unpooling": in a 2x2 maxpool, remember which position had max; during unpooling, place element into that position
 - Learnable upsampling - "**transposed convolution**"
 - Maps single pixel in input to larger kernel (3x3, e.g.) in output
 - Move one pixel in input -> 2 pixels in output, e.g.
 - Sum where outputs overlap
 - *Intuition:* can express convolution as matrix multiplication -> transposed convolution is multiplication by inverse matrix

Encoder-decoder networks

- Want to improve encoding of spatial information (encoder), maintaining spatial structure of mask (decoder)
- Encoder

- ***Dilated/astrous convolution*** - rather than multiplying a contiguous region of input image during convolution (3x3 kernel -> 3x3 region, e.g.), place gaps of 1 pixel between each sampled input region (3x3 kernel -> 5x5 region, e.g.)
 - Creates larger receptive field
- Feature pyramid structures
 - *Pyramid scene parsing network/PSPnet*: use multiple differently-sized convolutions on the same set of features and fuse outputs
 - Hypothesis: different sizes encode different kinds of features
 - *Feature Pyramid Networks/FPN*: predict at each feature size (?)
- **U-Net** - popular network for biomedical image segmentation

Semantic segmentation datasets - Cityscapes, MIT ADE20K

- ADE20K - manually labeled over several years by a single expert annotator

Instance Segmentation

Instance segmentation: detect all objects in images + corresponding pixels

One approach - perform object detection, then predict segmentation mask for each object

- **Mask R-CNN:** Uses Faster R-CNN approach + adds segmentation mask prediction to outputs
 - Outputs: object category, bounding box, segmentation mask

Instance vs semantic segmentation

- Instance segmentation - labels pixels + detects individual instances, but only for pixels corresponding to detected objects (not for background, sky, e.g.)
- Semantic segmentation - labels all pixels, but not instances

More advanced CV tasks

- **Panoptic segmentation** - label all pixels in image & separate object instances (for objects)
 - Combination of instance, semantic segmentation
- **Human keypoints** - detect human pose via locating a set of keypoints
 - Can extend Mask R-CNN, add keypoint positions as additional output
 - In addition to segmentation mask, output individual masks for each keypoint (K many)
 - Ground truth - one "pixel" enabled per keypoint, use softmax loss

General approach to advanced CV tasks: Add additional "heads"/outputs to Mask R-CNN

- Heads incorporated into per-region network
- Ex:
 - LSTM head for dense captioning
 - Mesh R-CNN: mesh predictor head

Segment Anything (SAM): foundation model for image segmentation

Generative Models

Supervised vs unsupervised learning

- **Supervised learning:** Given a set of data (x, y) , want to learn function mapping x to y
 - Ex: image classification, object detection, semantic segmentation
 - *Intuition:* Attempts to learn conditional probability distributions $p(y | x)$
 - Cannot sample raw distribution $p(x)$
- **Unsupervised learning:** Given a set of data x (no labels), want to learn some underlying structure of the data
 - Ex: clustering, dimensionality reduction, feature learning, density estimation
 - Attempts to learn unconditional probability distribution $p(x)$
 - Allows for sampling from $p(x)$ directly

Computer vision

- Supervised learning via **discriminative models**
 - For classification, segmentation, etc.
 - Lots of success across many tasks
- **Generative models** - unsupervised learning
 - More recent interest nowadays
 - Many advances in recent years

Recall: probability mass of a given element in a distribution assigned by density function

- Discriminative models (learn $p(y | x)$): given an image, different labels compete for probability mass
 - Different images do not compete
 - Correct labels should be assigned more mass
 - Done via feature learning with labels
- Generative models (learn $p(x)$): different images compete for probability mass
 - "Reasonable" outputs should be assigned more mass
 - Trained via feature learning without labels
 - Sample to generate new data
- Conditional generative models - learn $p(x | y)$
 - Given a label, different images compete for probability mass

Classes of Generative Models

1. *Explicit density*: model can compute $p(x)$ explicitly
 - a. Tractable density - can compute $p(x)$ exactly
 - i. Ex: autoregressive, NADE/MADE, etc.
 - b. Approximate density - can only approximate $p(x)$
 - i. Variational: Variational autoencoder (VAE)
 - ii. Markov chain: Boltzmann machine, diffusion model
2. *Implicit density*: model does not explicitly compute $p(x)$, only samples from it
 - a. Markov chain: GSN
 - b. Direct: Generative adversarial networks (GANs)

Autoregressive Models

Goal: Want to write an explicit function $p(x) = f(x, W)$

- Given dataset $x^{(1)}, \dots, x^{(N)}$, train the model to solve $W^* = \arg \max_W \prod_i^N p(x^{(i)})$
- Loss function: $\arg \max_W \sum_i \log f(x^{(i)}, W)$
 - Maximum likelihood estimation
- Idea: for multi-part inputs $x = (x_1, \dots, x_T)$, model using conditional probabilities
$$p(x) = \prod_i^T p(x_i | x_1, \dots, x_{i-1})$$

PixelRNN - generates pixels one at a time, starting from upper left corner

- Via RNN - computes hidden state for each pixel based on hidden states and RGB values from all preceding pixels (pixels directly to the left and above)
 - At each pixel, predict R, G, B separately and softmax over $[0, 1, \dots, 255]$
 - Recurrences via LSTM
- Issue: slow to train & test (requires $2N - 1$ sequential steps)

PixelCNN - dependency on previous pixels modeled via CNN over some context region

- Faster to train than PixelRNN (convolution parallelizable), but still slow to generate due to sequential steps

Autoregressive Models

Pros:

- Explicitly computes likelihood $p(x)$
- Explicit likelihood of training data gives a good evaluation metric
- Good samples

Cons:

- Sequential generation is slow

Variational Autoencoders

Variational autoencoders (VAE) - define an intractable density (cannot compute or optimize; only directly optimize a lower bound on density)

Regular/non-variational autoencoders: unsupervised method for learning feature vectors from raw data without labels

- Originally linear + nonlinearity; later MLP, ReLU CNN
- Want to learn useful features from data for downstream tasks; Q: how?
 - Idea: Train model as two parts: **encoder & decoder**
 - **Decoder** attempts to reconstruct original input data from encoded features
 - Loss: L2 between original, reconstructed features
 - Ex: conv layers in encoder -> transpose conv in decoder
 - **Encoder** portion is used as autoencoder
 - Throw away decoder after training -> encoder for downstream tasks
- Can use encoder to initialize a supervised model
 - From there, use to train on a final task (with limited data, e.g.)
 - Ex: dimensionality reduction with unsupervised learning
- Regular autoencoders not probabilistic - no way to sample new data from learned model

Variational autoencoders - probabilistic variation on regular autoencoders

- Learn latent features z from raw data -> can sample from model to generate new data
- Assumes training data $x^{(1)}, \dots, x^{(N)}$ generated from unobserved/latent representation z
 - Intuition: x an image; z latent features used to represent image (e.g. attributes, orientation, etc.)
- After training: first sample z from prior $p_{\theta^*}(z)$, then sample x from conditional distribution $p_{\theta^*}(x|z)$
 - Assumes simple prior $p_{\theta^*}(z)$ (Gaussian, e.g.)
 - Represent $p_{\theta^*}(x|z)$ via NN (similar to autoencoder decoder network)
 - Want to sample x from Gaussian with mean $\mu_{x|z}$, diagonal covariance $\Sigma_{x|z}$
 - Diagonal prior causes dimensions of z to be independent

- Idea: want to learn $p_\theta(x) = \frac{p_\theta(x|z)p_\theta(z)}{p_\theta(z|x)}$
 - $p_\theta(x|z)$ via decoder network; from Gaussian prior
 - $p_\theta(z|x)$: train another network (encoder) $q_\phi(z|x)$ to learn this
 - Want $q_\phi(z|x) \approx p_\theta(z|x)$

Network:

- Encoder network inputs data x , gives distribution over latent codes z (learns $\mu_{z|x}, \Sigma_{z|x}$)
- Decoder network inputs latent code z , gives distribution over data x (learns $\mu_{x|z}, \Sigma_{x|z}$)
- Want to jointly train both encoder, decoder
 - Train to maximize variational lower bound on data likelihood

$$\log p_\theta(x) \geq E_{z \sim q_\phi(z|x)}[\log p_\theta(x|z)] - D_{KL}(q_\phi(z|x), p(z))$$

Fully-connected VAE:

- Each of encoder, decoder has one initial linear layer + two parallel linear layers for μ, Σ
 - Encoder: high-dimensional x to low-dimensional μ, Σ
 - Decoder: low-dimensional z to high-dimensional μ, Σ [approx. same dim as original x]

Training a VAE (to minimize $D_{KL}(q_\phi(z|x), p(z))$)

- Run input data through encoder to get distribution over latent codes
 - Want encoder output to match prior $p(z)$
 - Has closed-form solution if q_ϕ diagonal Gaussian, p unit Gaussian
- Sample latent code z from encoder output
- Run sampled code through decoder to get a distribution over data samples
 - Want original input data to be likely under encoder-output distribution - can sample a reconstruction

Sampling a VAE

- Sample z from prior $p(z)$
- Run sampled z through decoder to get distribution over data x
- Sample from distribution to generate data

Can edit images after training:

- Run input data through encoder to get distribution over latent codes
- Sample code z from encoder output
 - Modify elements of sampled code as needed
 - Can use to change attributes of image, e.g.
- Run modified z through decoder to get new distribution & sample

Variational autoencoders

Pros:

- Principled approach to generative models
- Allows inference of latent codes $q(z|x)$ -> can be used as feature representation elsewhere

Cons:

- Maximizes lower bound of likelihood rather than computing distribution directly
- Samples are relatively blurry/low-quality compared to SoTA

Vector-quantized VAE (VQ-VAE) - combines VAE and autoregressive models

- VAE-like encoder generates latent space
- Converts continuous latent space into discrete distribution (latent code) via vector quantization
 - Trains an autoregressive model on discrete distribution as decoder

Training VQ-VAE

- Train a VAE-like encoder-decoder model to generate multiscale grids of latent codes from input data
 - Decoder - train to reconstruct from latent code
- Use multiscale PixelCNN to sample in/generate from latent code space

VQ-VAE improves on VAE in terms of image quality

- Used as image generation backbone in DALL-E (translating text to images) - uses VQ-VAE to decode text embedding space to generate images

Generative Adversarial Networks (GANs)

Generative adversarial networks (GANs) - train two separate networks: a **generator network** G and a **discriminator network** D

- Assume have data x_i from $p_{data}(x)$, want to sample from p_{data}
 - Idea: introduce latent variable z with simple prior $p(z)$
 - Sample $z \sim p(z)$ and pass to generator network $x = G(z)$
 - Take samples x from generator distribution p_G (want: $p_G = p_{data}$)
 - Train G to “fool” discriminator D
- Discriminator is a classifier network, train to classify data as real/fake
 - Train generator network G to generate an image (taking a sample from generator distribution) and fool the discriminator

Training GANs

- Discriminator, generator trained jointly via *minimax* objective
 - Minimax - generator trains to minimize (maximum error across all discriminators)
 - G, D share same loss function, but opposite objectives
 - $\min_G \max_D V(D, G)$ [note: no overall loss]
- Doesn't explicitly model $p(x)$, only samples from it
- Training GANs is difficult, unstable training + loss
 - Plot $\log(1-D(G(z)))$
 - Generator initially very bad (very easy for discriminator to distinguish - $D(G(z))$ near 0), improves over time
 - Issue: vanishing gradients for 0 when $D(G(z))$ near 0 [$\log(1-D(G(z)))$ small]
 - Solution: train G to minimize $-\log(D(G(z)))$
- GANs provably achieve global min when $p_G = p_{data}$
 - No guarantees on convergence of G, D to optimal

Improving GANs: better loss functions, StyleGAN for higher resolution

- La begin with fixed constant input vector & add in new layer-wise random latent vectors at each layer

GANs - latent space actually encodes semantic information; can identify patterns in latent space based on semantic attributes (e.g. "man")

- Latent space is continuous - can perform random walk to traverse
- Can identify subspaces associated with causal relations in latent space using unsupervised learning
 - Can use for manipulation

Image-to-image translation (P2P) - can use GANs to translate image types between different domains

- Generator takes input image (rather than random noise) as input; discriminator takes both input image, generator-output image
- *Issue:* for training, need image pairs (one in each domain) for paired translation
 - **CycleGAN** - take two sets of images (one in each domain) with no pairing
 - **Cycle reconstruction loss** - minimize reconstruction error from converting from one domain to the other & back again
- Ex: road map to fake satellite image

Diffusion Models

Diffusion models

- Training: rather than one-shot (GAN/VAE), train a model to gradually add Gaussian noise (encoder - forward/diffusion process) and then reverses/denoises the noise (decoder)
 - Decoder uses noise from encoding process to denoise
 - Markov chain process: each step depends only on output of previous step
 - After training: can sample white noise and pass to decoder to generate an image

Diffusion - Forward/diffusion process

- At each time step, at some amount of noise ϵ_t from standard normal distribution
 - Hyperparameter β_t (noise schedule) determines rate of noise blending
 - Given initial density $p(x)$, diffusion process gradually blurs distribution (moves toward standard normal distribution)
- Take T many steps $x \rightarrow z_1 \rightarrow \dots \rightarrow z_T$
 - From input image x to z_T (approximately pure noise)

Diffusion - Reverse/denoising process

- Want to learn series of probabilistic mappings $z_T \rightarrow z_{T-1} \rightarrow \dots \rightarrow z_1 \rightarrow x$
 - Individual mappings: $p(z_{t-1}|z_t, \phi_t)$
 - Maps z_T (pure noise) back to input image x (during training)
 - Via learned neural network
 - Pass in image + time embedding
- Overall, want to learn: $\hat{\phi}_{1...T} = \arg \max_{\phi_{1...T}} \left[\sum_{i=1}^I \log [p(x_1|\phi_{1...T})] \right]$

In practice: use diffusion encoder/decoder as encoder/decoder in U-Net model

- From noise image z_T , use U-Net as denoising network for each step $z_t \rightarrow z_{t-1}$
- After training: pass white noise image + noise concatenations as input to decoder

Latent diffusion: use encoder/decoder to move to/from pixel to latent space (similar to VAE)

- Use initial encoder to convert input x into latent space before performing forward diffusion; use final decoder to convert denoising output z back into pixel space output \tilde{x}

- Performs diffusion within latent space for efficiency + speed
- Ex: unCLIP model
 - Text encoder generates text embedding
 - Use text embedding as input to MLP generating latent code; latent code uses diffusion model as decoder (to generate images)
 - Training - use image encoder (CLIP, e.g.) to convert sampled image to image encoding; CLIP objective - match image encoding with original text encoding

Latent diffusion - extensions

- *Stable diffusion*
- Adding control to text-to-image diffusion (e.g. *ControlNet* - different visual inputs)
 - ControlNet - input condition image & text prompt together; incorporate image encodings of image into diffusion decoder stages via convolution
 - FreeControl: training-free control with any condition

Generative Models (Summary)

1. **Autoregressive models:** directly maximize likelihood of training data

$$p_{\theta}(x) = \prod_{i=1}^N p_{\theta}(x_i | x_1, \dots, x_{i-1})$$
 - a. Good quality, but slow & hard to scale
2. **Variational autoencoders:** introduce latent z for interpolation/editing
 - a. Maximizes lower bound
3. **Generative adversarial networks:** don't model $p(x)$, but samples $p(x)$
 - a. Good qualitative results
4. **Diffusion models:** use long Markov chain of diffusion steps to model $p(x)$
 - a. Flexible, but expensive to evaluate/train/sample

Trends in CV

Current trends:

- Ultra-large vision via foundation models
 - Scaling to large image datasets (OpenAI CLIP)
 - Multimodal image understanding models (LLaVA)
 - Image segmentation (SAMs)
 - Text2Image/Text2Video generation (Stable diffusion, Mochi)
 - Recognition & generation (Chameleon, Janus)
- 3D vision from multiple cameras & neural rendering
 - 3D perception with more cameras/sensors
 - 3D scanning
 - Recognizing 3D shapes (Mesh R-CNN, 3D object detection)
 - Interactive environments for embodied AI
 - Neural rendering: NeRF and Gaussian splatting for surface reconstruction & novel view synthesis from sets of images of objects/environments
 - 3D Gaussian splatting for real-time radiance fields

Challenges

- Interpretability, safety, robustness, etc.
 - Interpretability of AI model
 - AI safety in real-world applications
 - CV models fragile, easily fooled
 - Bias in visual classifiers, datasets
- Need a large amount of training data
 - Low-shot learning - learning from small datasets
 - Self-supervised learning - learning from unlabeled data