Low Level / Classical Vision

Explain/draw out a sketch of the pinhole camera model.

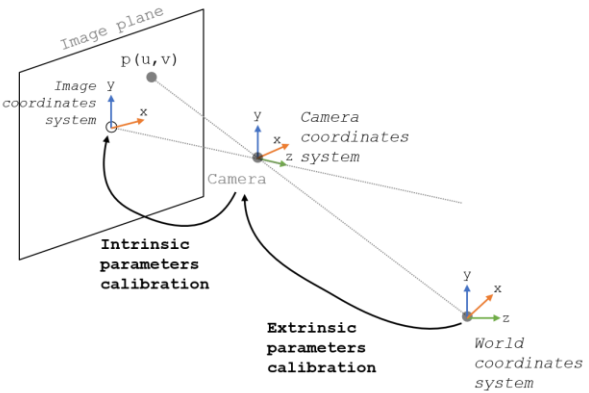
$$[R|t] = \begin{bmatrix} R_{3\times3} & T_{3\times1} \\ 0_{1\times3} & 1 \end{bmatrix}_{4\times4}$$

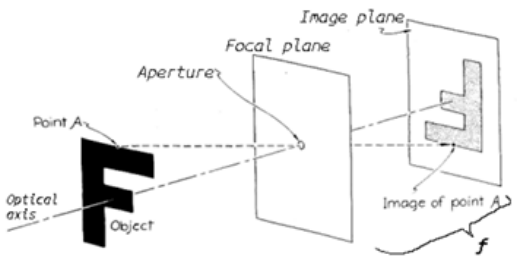
$$\begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} R_c & C \\ 0 & 1 \end{bmatrix}^{-1}$$
$$= \left[\begin{bmatrix} I & C \\ 0 & 1 \end{bmatrix} \begin{bmatrix} R_c & 0 \\ 0 & 1 \end{bmatrix} \right]^{-1}$$
$$= \begin{bmatrix} R_c & 0 \\ 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} I & C \\ 0 & 1 \end{bmatrix}^{-1}$$
$$= \begin{bmatrix} R_c^T & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} I & -C \\ 0 & 1 \end{bmatrix}$$
$$= \begin{bmatrix} R_c^T & -R_c^T C \\ 0 & 1 \end{bmatrix}$$

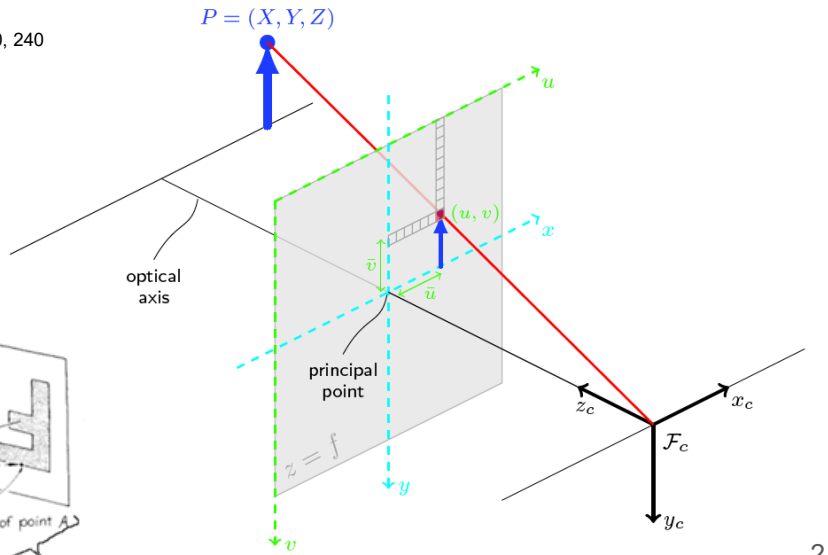
- The pinhole camera model describes a simplified way that a point in 3D is projected onto a 2D image plane.
- The **extrinsic camera matrix** involves the 3D location of the camera.
  - It is a matrix that takes world coordinates and converts them into camera coordinates
  - R is a 3x3 matrix where the columns represent the world axes in camera coordinates (world_to_cam)
  - T is a 3x1 vector representing the world origin in camera coordinates
  - R and T is a bit unintuitive to specify, so often we first obtain:
    - R_c ,a 3x3 matrix where columns represent the camera axes in world coordinates (cam_to_world)
    - C, a 3x1 vector representing the location of the camera center in world coordinates
    - Then, we can get:
    - $R = R_c^T$
    - $T = -R_c^T C$

$$K = \begin{bmatrix} f_x & s & x_0 \\ 0 & f_y & y_0 \\ 0 & 0 & 1 \end{bmatrix}$$

- The **intrinsic camera matrix** K represents the internal parameters of the camera, and can be found using camera calibration:
  - focal length f_x, f_y, the distance between the pinhole and the image plane in pixels. Affects size of object
  - skew s. Shifts objects at an angle, typically zero
  - principal point offset, x_0, y_0 , affects xy location of object. Eg for a 640 x 480 camera, this would generally be 320, 240







What are the equations to turn a 3D point to 2D, and a 2D point to a 3D ray?

3D point -> 2D point (Projection)

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \boldsymbol{K} \frac{1}{Z} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{R} & \boldsymbol{t} \\ \boldsymbol{0} & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

2D point: $\begin{bmatrix} \frac{x}{w} \\ \frac{y}{w} \end{bmatrix} = \begin{bmatrix} u \\ v \end{bmatrix}$

(Note, when multiplied out the rotation happens before the translation since the latter is based on the camera frame)

2D point -> 3D Ray (Unprojection)

Direction of ray in camera coordinates: $\lambda K^{-1} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}$

(λ scales the ray)

Transformed to world coordinates:

$$R^T \left(\lambda K^{-1} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} - t \right) = \lambda R^T K^{-1} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} - R^T t$$

(Here, we undo the rotation/translation in the opposite order as before)

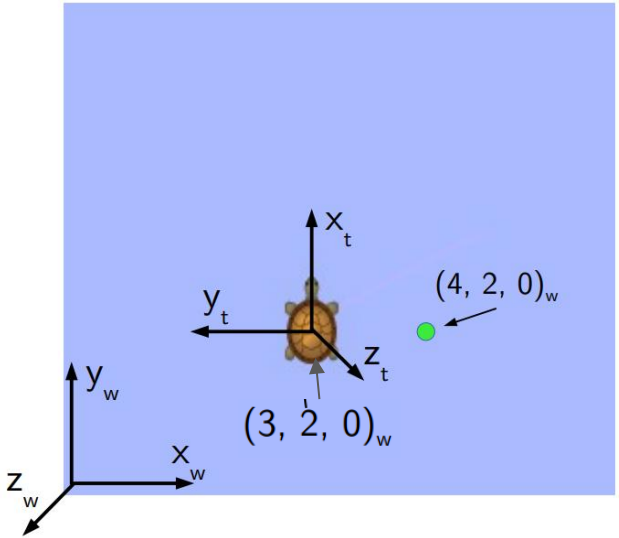
What is (4,2,0)w in turtle coordinates?

$$R_{t2w} = \begin{bmatrix} 0 & -1 & 0 & 3 \\ 1 & 0 & 0 & 2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T = -\begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 3 \\ 2 \\ 0 \end{bmatrix} = \begin{bmatrix} -2 \\ 3 \\ 0 \end{bmatrix}$$

$$R_{w2t} = \begin{bmatrix} 0 & 1 & 0 & -2 \\ -1 & 0 & 0 & 3 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_{w2t}\begin{bmatrix} 0 \\ -1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ -1 \\ 0 \\ 1 \end{bmatrix}$$



What are 2 types of camera distortion and how can it be corrected?

- Can use a library like opencv to find distortion coefficients given multiple checkerboard imgs
- chessboard is great for calibration because its regular, high contrast pattern makes it easy to detect automatically. And we know how an undistorted flat chessboard looks like

Radial

$$x_{distorted} = x(1 + k_1*r^2 + k_2*r^4 + k_3*r^6)$$

$$y_{distorted} = y(1 + k_1*r^2 + k_2*r^4 + k_3*r^6)$$

- x, y — Undistorted pixel locations. x and y are in normalized image coordinates. Normalized image coordinates are calculated from pixel coordinates by translating to the optical center and dividing by the focal length in pixels. Thus, x and y are dimensionless.
- k_1, k_2, and k_3 — Radial distortion coefficients of the lens.
- $r^2 = x^2 + y^2$

Tangential

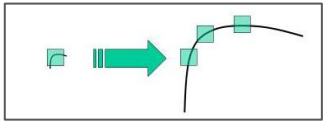
$$x_{distorted} = x + [2 * p_1 * x * y + p_2 * (r^2 + 2 * x^2)]$$

$$y_{distorted} = y + [p_1 * (r^2 + 2 * y^2) + 2 * p_2 * x * y]$$

- x, y — Undistorted pixel locations. x and y are in normalized image coordinates. Normalized image coordinates are calculated from pixel coordinates by translating to the optical center and dividing by the focal length in pixels. Thus, x and y are dimensionless.
- p_1 and p_2 — Tangential distortion coefficients of the lens.
- $r^2 = x^2 + y^2$

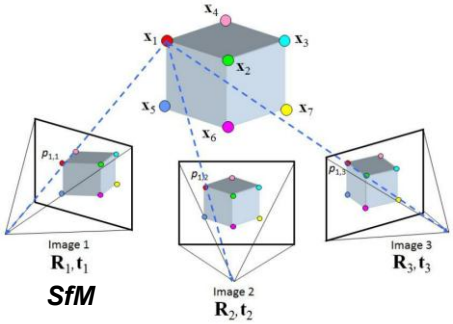
Left/middle are radial distortion, right is tangential distortion.

What is SIFT and what problems does it tackle?

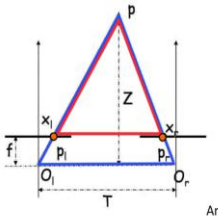


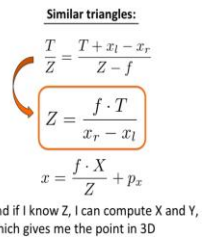
- The **Scale-Invariant Feature Transform** (1999) is a feature detection algorithm which is scale and rotation invariant.
  - It is an improvement of things like the **Harris corner detector** (1988), which is only rotation invariant (since, corners look like lines when zoomed in)
- Once you obtain SIFT keypoints for one image, you can recognize objects in other images by getting its SIFT keypoints and comparing them to the original keypoints with euclidean distance of the feature vectors, thus establishing **correspondences**. In general, the process of matching features is called **image registration**.
- Downstream uses include:
  - **Object recognition** (if number of high confidence correspondences exceed a threshold)
  - **Robot localization** (by feature mapping to a known 3D map)
  - **Depth estimation** (By using SIFT to generate correspondences between stereo images)
  - **Structure from motion** (By using SIFT to generate correspondences between many images; if camera parameters are known, we get the depth of an object and can generate a pointcloud or mesh)
  - **Panorama stitching/image alignment**
  - **Motion tracking** of an object
  - Feature-based **optical flow**

Image Stitching

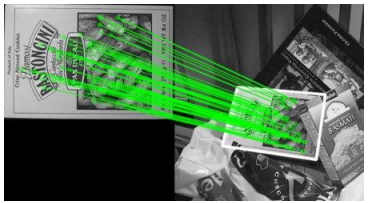


SfM





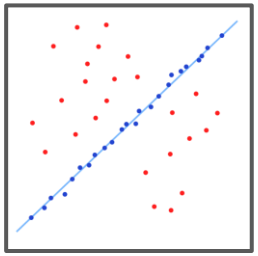
Depth Estimation



Correspondences

What is RANSAC and how can it be used to find correspondences?

Random sample consensus (RANSAC) is an iterative probabilistic method to estimate parameters of a mathematical model from a set of observed data that contains outliers. Here, outliers are not given any influence on the values of the estimates; only inliers.

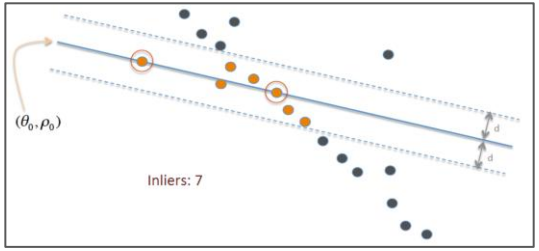


RANSAC psuedocode:

Input to RANSAC: set of observed data values, way of fitting some kind of parameterized model to the observations, and some hyperparameters

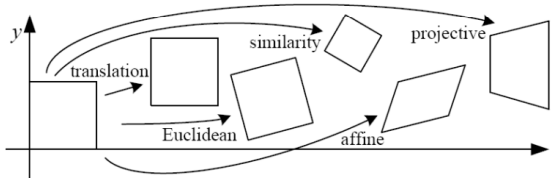
Until convergence:

- *Select a very small random subset of the original data. Call this subset the hypothetical inliers.*
- *A model is fitted to the set of hypothetical inliers.*
- *All other data are then tested against the fitted model. Those points that fit the estimated model well, according to some model-specific loss function, are considered as part of the consensus set.*
- *The estimated model is reasonably good if sufficiently many points have been classified as part of the consensus set.*
- *Afterwards, the model may be improved by reestimating it using all members of the consensus set.*



For example, **RANSAC can be applied to linear regression** to exclude outliers.

- It fits linear models to several very small random samplings of the data and returns the model that has the best fit to a subset of the data.
- Since the inliers tend to be more linearly related than a random mixture of inliers and outliers, a random subset that consists entirely of inliers will have the best model fit.
- In practice, there is no guarantee that a subset of inliers will be randomly sampled, and the probability of the algorithm succeeding depends on the proportion of inliers in the data as well as the choice of several algorithm parameters.



RANSAC can be applied to find correspondences between SIFT features in two images. In this case, the model is a **homography** (aka **projective transformation**) that attempts to align two images taken from different perspectives together.

- A homography is a isomorphism between two vector spaces; ie, representable by a nonsingular matrix.
- Homographies are viable either of the following are true:
  - The scene is planar or approximately planar (ie, the scene is very far or has small relative depth variation)
  - The scene is captured under camera rotation only (no translation or pose change)

Left is Planar; Right is Approx Planar due to distance; Bottom is captured under cam. Rotation only

At a high level to use RANSAC for SIFT correspondences, you repeatedly:

- Randomly pick 4 good matches (based on l2 distance); compute homography
- Check how many good matches are consistent with homography, to find hypothetical inliers/outliers

In the end, you keep the homography with the smallest number of outliers.

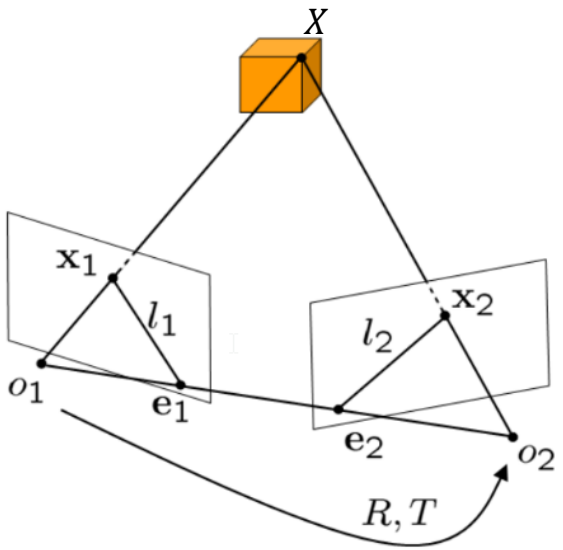
Describe the steps that go into a classical structure from motion pipeline, eg COLMAP for Offline, ORB-SLAM or Online.

- **Image Acquisition**
  - Want set of overlapping images from different viewpoints
- **Feature Detection/Description**
  - Detect keypoints and extract invariant/robust features, eg SIFT, SURF, ORB, etc
- **Feature Matching**
  - Match descriptors, eg using nearest neighbors & other heuristics
- **Initial Pair Selection & Pose Estimation**
  - Choose initial image pair with a good number of robust matches & sufficient baseline
  - Estimate Essential Matrix (if intrinsics known), else Fundamental Matrix using the **eight-point-algorithm**
  - Essential/Fundamental matrix can be decomposed to obtain relative rotation & translation between the cameras
  - Triangulate matched features to get initial 3D points
- **Iterative Structure & Motion Recovery**
  - Select new Image with sufficient feature matches to already constructed 3D points
  - Estimate camera pose using **PnP (Perspective-n-Point)**
    - Here we use existing found 3D points and their 2D correspondences in the new image, to compute the pose for the new image
  - Triangulate new 3D points with the new image using the recovered pose
- **Global Bundle Adjustment**
  - Non-linear optimization that minimizes the reprojection error across all images, camera poses, and 3D points
  - Given m images, n 3D points, and mn 2D point coordinates $\widetilde{x_i^j}$, want to minimize

    $E\left(\{R_i, T_i\}_{i=1,\dots,m}, \{X_j\}_{j=1,\dots,n}\right) = \sum_{i=1}^{m}\sum_{j=1}^{n}\theta_{ij}\left|\widetilde{x_i^j} - \pi(R_i, T_i X_j)\right|^2$

    Where $\theta_{ij} = 1$ if point j is visible in image i, else 0 and π denotes the perspective projection function
  - Typically solved using Levenberg-Marquardt or gradient descent
- **Loop closure (Optional)**
  - Detect when the camera revisits a previously seen location, and do a pose graph optimization

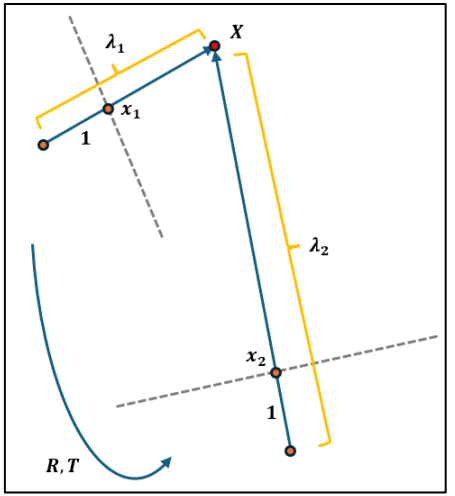
Derive the epipolar constraint & describe its geometric interpretations.



- The right figure illustrates the geometry of a 3D point & 2 cameras:
  - o_1, o_2: **Optical centers** of each camera
  - R, T: relative rotation & translation between the cameras
  - x_1, x_2: **Projections** of 3D point X onto two images.
    - In normalized image coordinates, ie in camera frame after applying intrinsics with K^{-1}
  - e_1, e_2: **Epipoles**, which are the intersection of the line (o_1, o_2) with each image plane
  - (o_1, o_2, X): **Epipolar plane**; there is one such plane for each 3D point X
  - (o_1, o_2): **Baseline vector** between the two cameras
  - l_1, l_2: **Epipolar lines**: intersections between the image planes & the Epipolar plane. Note that line (o_1, X) projects to l_2, & x_2 must lie on l_2. Similarly line (o_2, X) projects to l_1, & x_1 must lie on l_1.
- Recall the following properties of skew symmetric matrices $so(3)$:
  - $M \in so(3) \leftrightarrow M^T = -M$
  - Has isomorphism with $\mathbb{R}^3$ via the "hat map":

$$\hat{u} = \begin{pmatrix} 0 & -u_3 & u_2 \\ u_3 & 0 & -u_1 \\ -u_2 & u_1 & 0 \end{pmatrix}$$

  - For any vector v, $u \times v = \hat{u}v$; so this is another way to express cross products
- Then, the Epipolar constraint can be derived as follows:
  - $\lambda_1 x_1 = X, \quad \lambda_2 x_2 = RX + T$
  - $\lambda_2 x_2 = R(\lambda_1 x_1) + T$
  - $\lambda_2 \hat{T} x_2 = \lambda_1 \hat{T} R x_1$, since $\hat{T}T = 0$
  - $x_2^T \hat{T} R x_1 = 0$, since x_2 is orthogonal to $\hat{T}x_2 = T \times x_2$
  - $x_2^T E x_1 = 0$, where E is the essential matrix
- Geometric interpretations:
  - Enforces that the three vectors x_2, T, and Rx_1 indeed form a plane, i.e. the triple product forms a zero-volume parallelepiped
    - This is expressed in camera 2's coordinate frame. T starts from the origin and ends at o_1 representing the baseline; Rx_1 is x_1 in camera 2's frame, where we can ignore the translation since we're working with vectors.
  - Says that x_2 must lie on Epipolar line $l_2 = Ex_1$, and similarly that that x_1 must lie on Epipolar line $l_1 = x_2^T E$
    - Recall that in homogenous coordinates, a vector $\begin{bmatrix} a \\ b \\ c \end{bmatrix}$ represents a line; when multiplied by coordinates $\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$ we get the equation of a line $ax + by + c = 0$



How is R, T estimated using the eight-point algorithm?

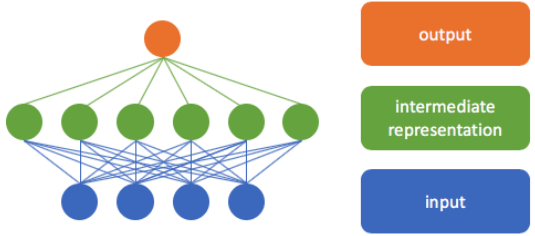
- Form a linear system $x_2^i E x_1^i = 0$ using the Epipolar constraint, with at least $n = 8$ corresponding pairs of points
- Stack into system $Ae = 0$ where e is the vectorized essential matrix
  - $A = (a_1, \dots, a_n)^T$ is a $n \times 9$ matrix where $a^i = x_1^i \otimes x_2^i = \left(x_1^i x_2^i, x_1^i y_2^i, x_1^i z_2^i, y_1^i x_2^i, y_1^i y_2^i, y_1^i z_2^i, z_1^i x_2^i, z_1^i y_2^i, z_1^i z_2^i\right) \in \mathbb{R}^9$ is the Kronecker product.
- Minimize by performing SVD to obtain $A = U\Sigma V^T$, selecting the 9th column of V and unstacking to obtain E
  - Recall that solution lies in the nullspace of A; columns of V corresponding to zero singular values form a basis for the nullspace of A
  - In practice we might not get an exact zero singular value, but we can choose the smallest as an approximation
- The E obtained is not guaranteed to have rank 2 & be an essential matrix, so we project it onto the essential space
  - This is by computing $\mathrm{E} = U\, diag\{\sigma_1, \sigma_2, \sigma_3\}\, V^T$ and replacing with $U\, diag\{1,1,0\}\, V^T$
  - Essential matrices always have rank 2 since it is the result of a skew symmetric matrix (rank 2) and a rotation matrix (full rank)
- RANSAC can be used to choose the best E (most inliers)
  - Randomly sample minimal subsets of $n = 8$
  - Compute E
  - Count number of inliers by checking how close the other points are to solving the epipolar constraint

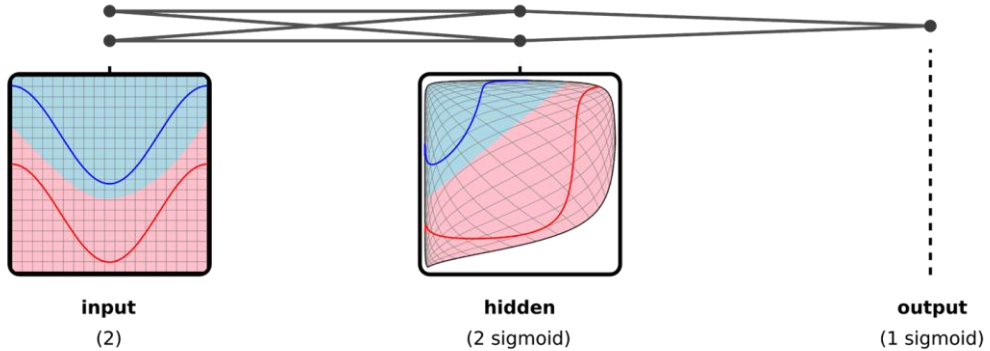
- **Obtain R,T from the Essential Matrix:**
  - There are four possible solutions for rotation & translation:
  - $R = UR_Z^\top(\pm\frac{\pi}{2})V^\top, \quad \widehat{T} = UR_Z(\pm\frac{\pi}{2})\Sigma U^\top$
  - $R_Z^\top(\pm\frac{\pi}{2}) = \begin{pmatrix} 0 & \pm 1 & 0 \\ \mp 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$
  - As expected, R_z does not affect the Z axis
  - The correct one can generally be easily found since only one will have non-negative depth (in front of image plane)

Deep Learning Fundamentals

In general, how do neural networks operate?

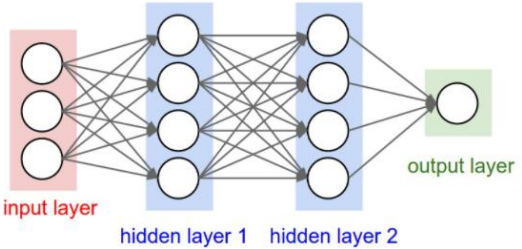
- Parametrized models which we optimize using a task-specific objective function, over a training dataset
- One perspective: takes inputs, transforms them into a useful internal feature representation which is linearly separable(i.e. feature extractor), and applies a linear (e.g. softmax) classifier at the end.

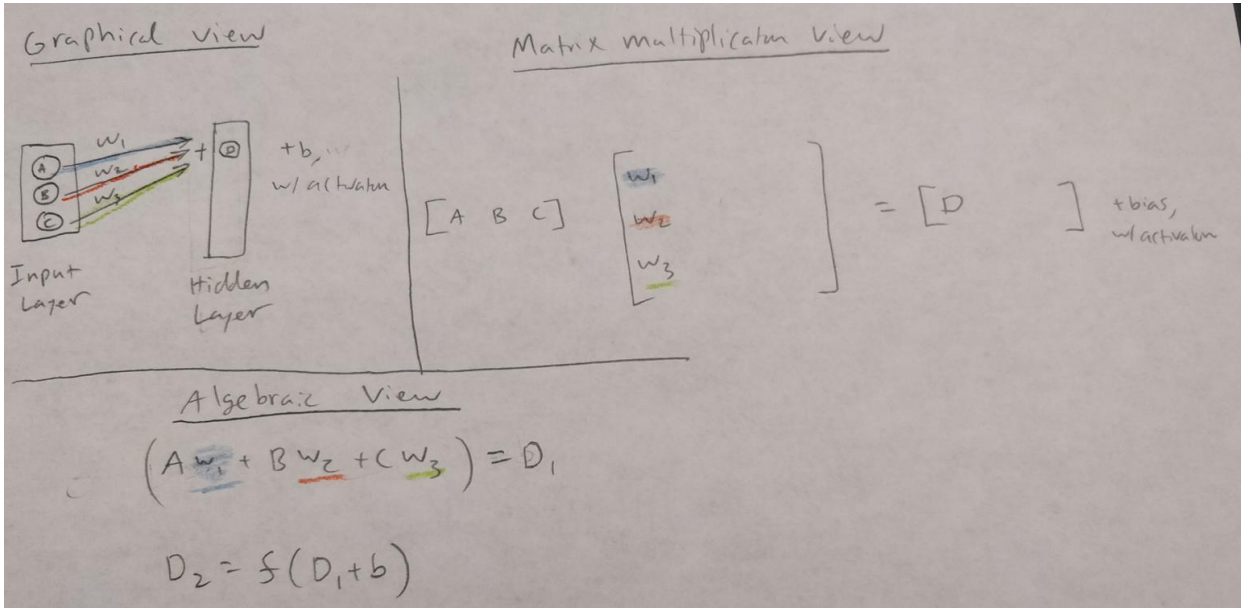


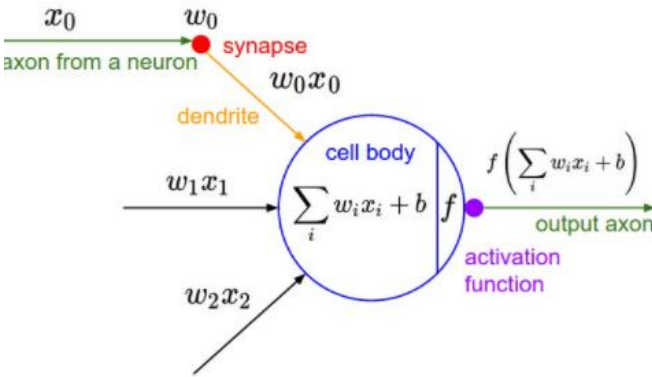


How many neurons & parameters does the below neural network have? Explain the Matrix multiplication view, graphical view, algebraic view, and biological view.

- Neurons: 9, not including inputs
- [3 x 4] + [4 x 4] + [4 x 1] = 12 + 16 + 4 = 32 weights and 4 + 4 + 1 = 9 biases, for a total of 41 learnable parameters.
- The 3 sets of arrows all represent an n*k matrix, where n is the number of neurons in the left layer and k is the number of neurons in the right layer.

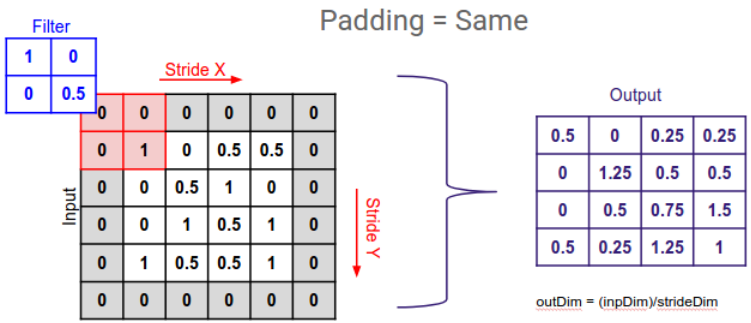






What is the output depth, padding, stride, and filter size of a conv layer?

- **Output depth** (O): equal to the number of filters used in the convolutional layer.
- **Padding** (P): The amount of zero padding before convolutions, which can help maintain the spatial size of the input/output volumes.
- **Stride** (S): The number of positions we move as we slide the filter around. For example, if this is 2, then the output volume is halved spatially (given sufficient padding)
- **Filter Size** (F): Determines how large the receptive field is
    - Also called the **kernel size**

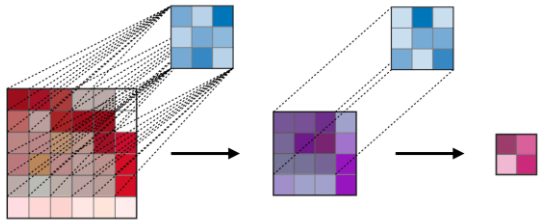


What makes convnets translationally invariant? What is the “receptive field”?

- Translational invariance occurs because each filter learns parameters which are the same spatially, and slid across the image.

❒ **Receptive field** — The receptive field at layer k is the area denoted $R_k \times R_k$ of the input that each pixel of the k-th activation map can 'see'. By calling F_j the filter size of layer j and S_i the stride value of layer i and with the convention $S_0 = 1$, the receptive field at layer k can be computed with the formula:

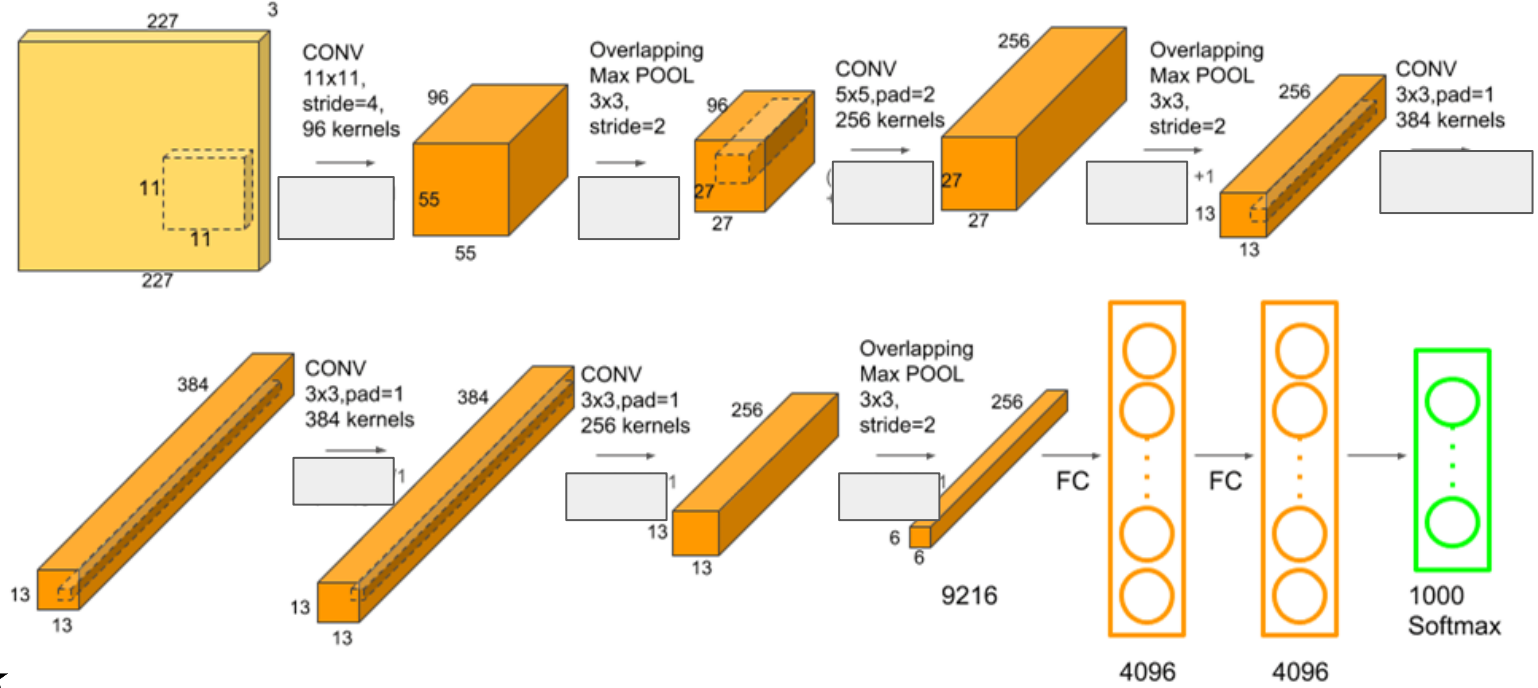
$$R_k = 1 + \sum_{j=1}^{k} (F_j - 1) \prod_{i=0}^{j-1} S_i$$



In the example below, we have $F_1 = F_2 = 3$ *and* $S_1 = S_2 = 1$*, which gives* $R_2 = 1 + 2 \cdot 1 + 2 \cdot 1 = 5$.

How do you calculate output volume size of a conv layer given input size W, padding P, stride S, filter (kernel) size F, and O filters? How do you maintain the spatial size for filter sizes 3 and 5?
What about a pooling layer of input size W, filter size F and stride S?

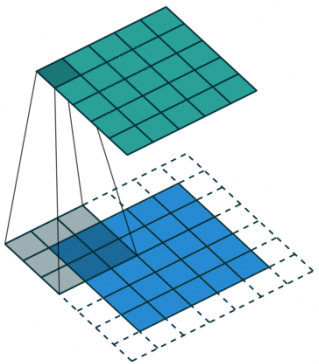
- Conv layer: Output spatial size will be D = ((W+2P-F)/S) + 1; output feature map size will be O x D x D
- Pooling layer: ((W-F)/S)+1; output depth stays the same
- To maintain spatial sizes:
  - For F=3, use P=1
  - For F=5, use P=2



Explain regular, pointwise/1x1, transpose/up/de/fractionally strided, atrous/dilated, and depthwise separable convolutions.

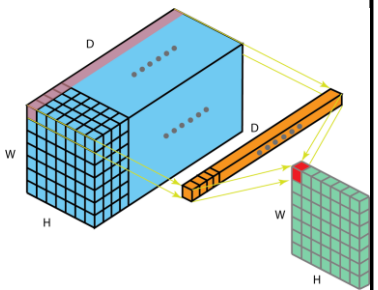
Regular Convolutions

- Example of a kernel/filter: n x n x c, where c is input feature map's channel size
- Dot products are computed, sliding across input feature map
- Each filter outputs 1 channel. For example, if we want our output to have 256 channels, we need 256 filters



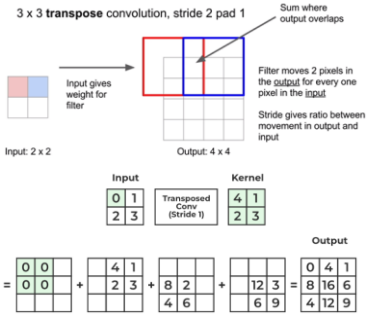
1x1 Convolutions

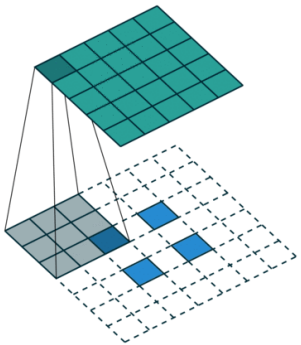
- Same as regular conv, but 1 x 1 x c filters. This keeps the spatial dimensions, but changes the channels
- Useful for dim. reduction, and introducing nonlinearities
- Output array from each filter after convolving with input can be thought of as a weighted sum of the input feature map's channels



De/Up/Transpose/fractionally strided Conv.

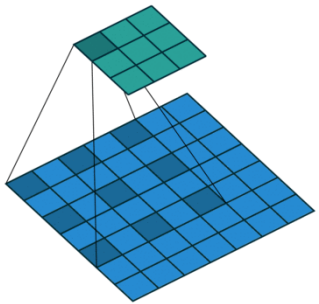
- Instead of sliding kernel over input & dot prod, input gives weight for filter and we sum where we overlap.





Atrous/Dilated Conv.

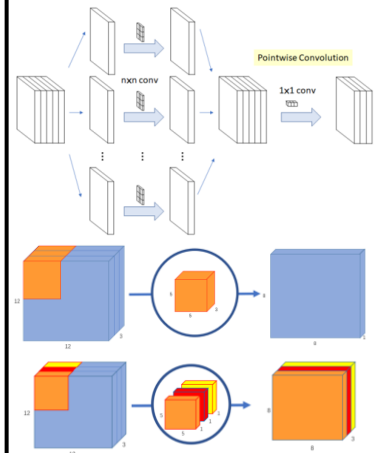
- Adds gaps in the filter when convolving, increasing the field of view even with the same number of parameters.
- However, this does not spatially upsample



Depthwise Separable Conv. (From MobileNet)

For efficiency, this operation decomposes the regular convolution into two stages:

1) A depthwise convolution operation, preserving the depth C of the input. C many "groups" (pytorch terminology) of n x n x 1 channel-specific kernels are used, each separately applied to each channel of the input to get C channels.
2) 1 x 1 pointwise convolutions are used, to get desired depth.



Normal (top) vs depthwise (bottom) conv

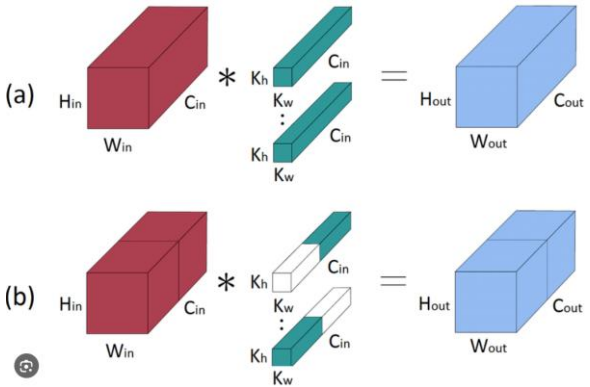
Derive the number of multiplications needed for depthwise separable vs regular convolutions, given an H x W input volume, a K x K filter, M channels in, and N channels out.

$$\frac{depthwise}{regular} = \frac{KKHWM + HWMN}{KKHWMN} = \frac{HWM(KK + N)}{KKHWMN} = \frac{KK + N}{KKN} = \frac{1}{N} + \frac{1}{K^2}$$

What are grouped convolutions?

`groups` controls the connections between inputs and outputs. `in_channels` and `out_channels` must both be divisible by `groups`. For example,

- At groups=1, all inputs are convolved to all outputs.
- At groups=2, the operation becomes equivalent to having two conv layers side by side, each seeing half the input channels and producing half the output channels, and both subsequently concatenated.
- At groups= `in_channels`, each input channel is convolved with its own set of filters (of size $\frac{\text{out\_channels}}{\text{in\_channels}}$).



a) Standard conv
b) Grouped conv

What are coord convolutions?

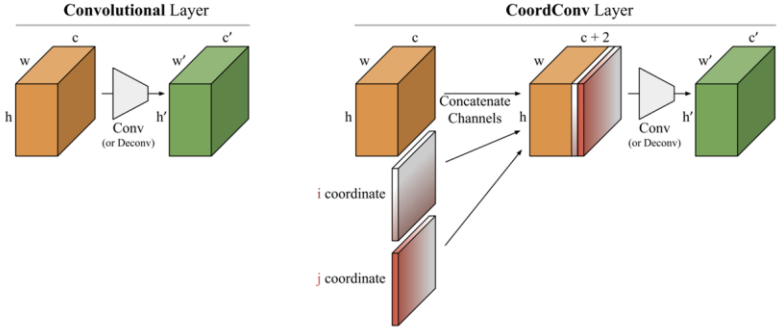
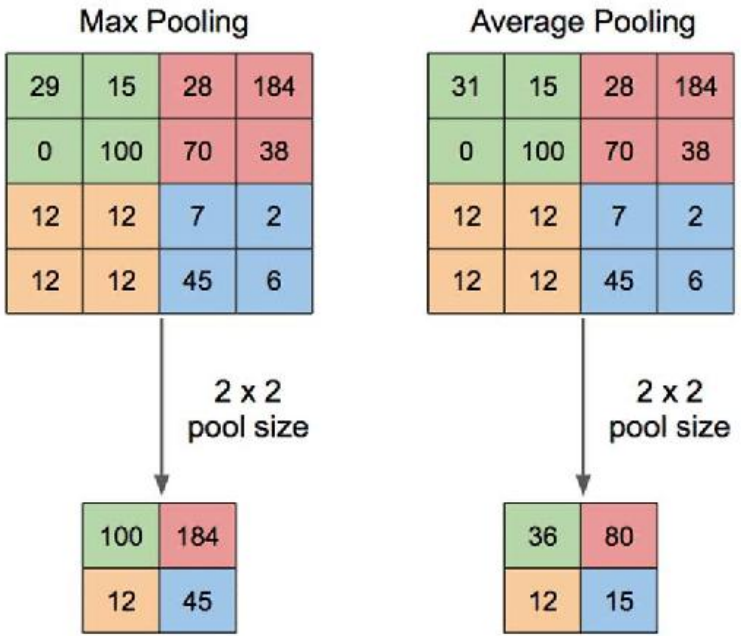


Figure 3: Comparison of 2D convolutional and CoordConv layers. **(left)** A standard convolutional layer maps from a representation block with shape $h \times w \times c$ to a new representation of shape $h' \times w' \times c'$. **(right)** A CoordConv layer has the same functional signature, but accomplishes the mapping by first concatenating extra channels to the incoming representation. These channels contain hard-coded coordinates, the most basic version of which is one channel for the i coordinate and one for the j coordinate, as shown above. Other derived coordinates may be input as well, like the radius coordinate used in ImageNet experiments (Section 5).

- While CNNs are powerful at learning hierarchical features, they don't inherently have built-in mechanisms for understanding the absolute or relative positions of objects within an image
- Adding coordinate channels provide the network with explicit spatial information, which can help it understand the location and orientation of objects in the input image
- Coordinate channels are typically computed as normalized values, meaning that the coordinates are scaled to a specific range (e.g., [-1, 1] or [0, 1]) to ensure that they don't dominate the other feature channels

How does max and average differ? Which is more popular? Why do some dislike pooling altogether?

- Max/average pooling just downsizes the spatial size, while keeping the feature dimension the same.
  - Both are applied separately for each of the input channels
- Some think that averaging all features in a very large region might just kill a lot of the information there
- Some people dislike the pooling operation and there has been work suggesting that we should avoid it in favor of convolutions that downsample by using larger stride once in a while. One example is the All-Convolutional Network paper (ICLR 15, 1500 cit.). This is also adopted in ResNet. This has been seen to be especially important in VAEs and GANs.



What is the softmax function, and what loss function is it usually associated with?

Given a vector of raw logits $\mathbf{z} \in R^K$, the softmax function $\sigma(\mathbf{z}) \in R^K$ such that

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for } j = 1, \ldots, K$$

- Intuitively, it is a smooth approximation to the arg max; it normalizes the input (called raw **logits**) into probabilities summing to 1.
- Example: $\sigma([1, 2, 3, 4, 1, 2, 3]^T) = [0.024, 0.064, 0.175, 0.475, 0.024, 0.064, 0.175]^T$
  - Softmax highlights the largest values and suppress values which are significantly below the maximum value

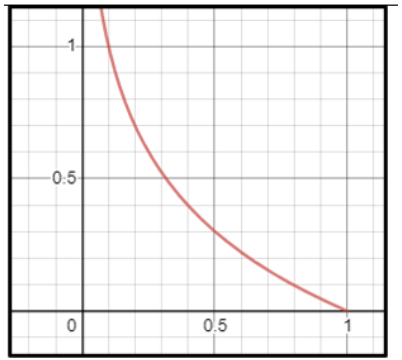
The softmax is often paired with the **cross-entropy loss** (aka **negative log likelihood**), which operates on vectors of class probabilities.

$$-\sum_{c=1}^{M} y_{o,c} \log(p_{o,c})$$

Note

- M - number of classes (dog, cat, fish)
- log - the natural log
- y - binary indicator (0 or 1) if class label c is the correct classification for observation o
- p - predicted probability observation o is of class c

$$f = -\log(x)$$



Draw and describe the following activation functions: sigmoid, ReLU, leaky ReLU, ReLU6, tanh,

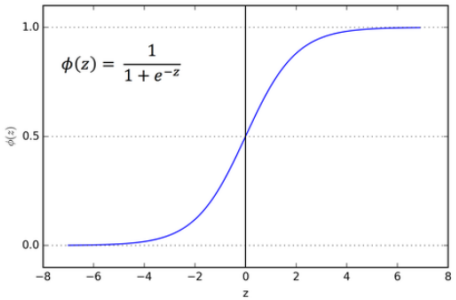


Fig: Sigmoid Function

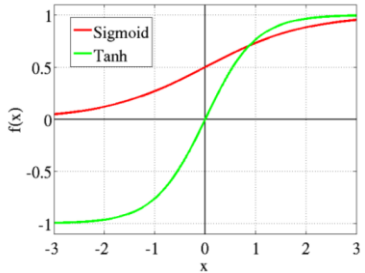
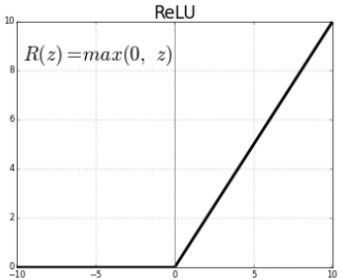
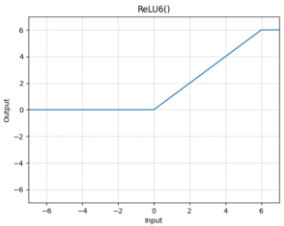
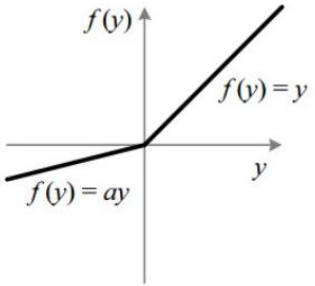


Fig: tanh v/s Logistic Sigmoid







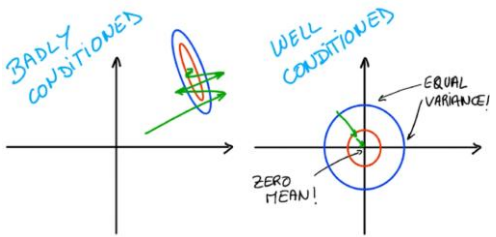
Leaky ReLU (usually a is around 0.01)

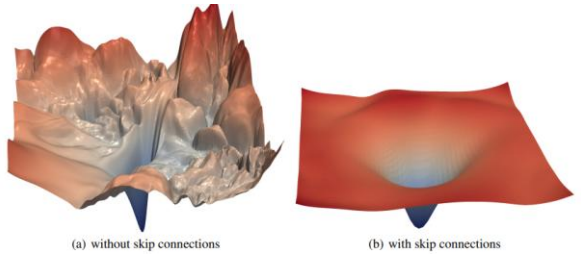
Some notes:

- Sigmoid and tanh can lead to vanishing gradients, so ReLU is most often used in the hidden layers
- In the binary case, sigmoid is equivalent to the softmax function where we just set one of the output nodes to be constant 0.
- Although ReLU is non-differentiable at 0, it's rare to have x = 0 exactly, and even it it is, we usually just hard-code it as 0, 1, or 0.5.
- Leaky ReLU avoids the issue of neurons being deactivated if negative.
- ReLU6 helps limit range for QAT precision

What is gradient descent?

- Given a function and an initial starting point on the domain of the function, **gradient descent** tries to find a local minima by repeatedly:
  - **Computing the gradient at the current position** (ie, calculate the derivative/direction of steepest ascent for each variable, which combines to form a vector)
  - Updating the current position: **subtracting the current position by the gradient times a small step size**
- In the context of neural networks:
  - The variables to optimize are the parameters of your neural network.
  - The function is called a loss function, and involves your training data. Usually, you sample a minibatch of the data.
    - Thus in minibatch gradient descent, **the loss function actually changes with each iteration**. Usually it changes in a linear way; terms in the summation of the loss function are left out depending on what examples are in the minibatch. This results in the loss function landscape (and consequently, gradient computed for the training step), being an approximation towards the overall loss function we actually wish to minimize, which incorporates all training examples. In practice, this approximation is worth it because it allows for more frequent updates.
- Loss function landscapes are highly non-convex, but regularizers can help improve the smoothness and prevent being stuck at saddle points too often
  - This is called "**conditioning**", and includes initialization, (batch/layer/etc) normalization, and residual connections





(a) without skip connections (b) with skip connections

Figure 1: The loss surfaces of ResNet-56 with/without skip connections. The proposed filter normalization scheme is used to enable comparisons of sharpness/flatness between the two figures.

At a high level describe SGD + momentum, adagrad, RMSprop, and Adam. Also explain (Multi)StepLR and reduceLROnPlateau.

- **SGD + Momentum** (Same learning rate applied to all parameters)
  - Adds the gradient at the last time step (times a momentum factor) with the current gradient.
  - The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions. This helps with faster convergence and less oscillation.

$$v_{t+1} = \mu * v_t + g_{t+1}$$
$$p_{t+1} = p_t - \text{lr} * v_{t+1}$$

- **AdaGrad** (Learning rate per parameter)
  - Divides the learning rate by the sum of squared gradients of each parameter up to the current timestep.
  - Parameters with past large gradients gets its learning rate progressively reduced more than those with smaller gradients.
  - Since we are squaring the gradients, the learning rate is monotonically decreasing in a quite aggressive way.

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

G_ii is the sum of squared gradients of theta_i up to time t.

- **AdaDelta/RMSprop** (Learning rate per parameter)
  - Learning rate is divided by the square root of a running weighted average of all past squared gradients for that parameter.
  - Similar to AdaGrad, params with previous large gradients get diminished more than those with smaller past gradients but less aggressive

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

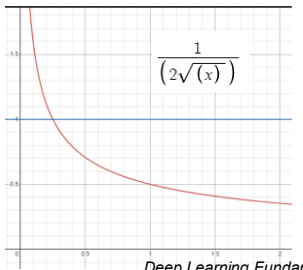
- **Adam** (Learning rate per parameter)
  - Keeps an (weighted) exponential moving average of the gradient and the squared gradient (ie, the first two moments)
  - The weighted gradient is used, and the learning rate is divided by the square root of the squared gradient.
  - Lower variance is more stable so will have higher learning rate; higher variance is unstable so will have lower learning rate

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$
$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

An additional benefit to note for the last three is that their **adaptive nature makes the optimization more robust to learning rate**.

Besides these, pytorch also has other schedulers, for example:

- torch.optim.lr_scheduler.**StepLR**
  - Decays learning rate by multiplication with *gamma* every *step_size* epochs.
- torch.optim.lr_scheduler.**MultiStepLR**
  - Decays learning rate by multiplication with *gamma* every time a specified milestone epoch is reached.
- torch.optim.lr_scheduler.**ReduceLROnPlateau**
  - Reads a metric quantity and if no improvement (up to a *threshold*) is seen for a *patience* number of epochs, the learning rate is reduced by multiplication with *gamma*.

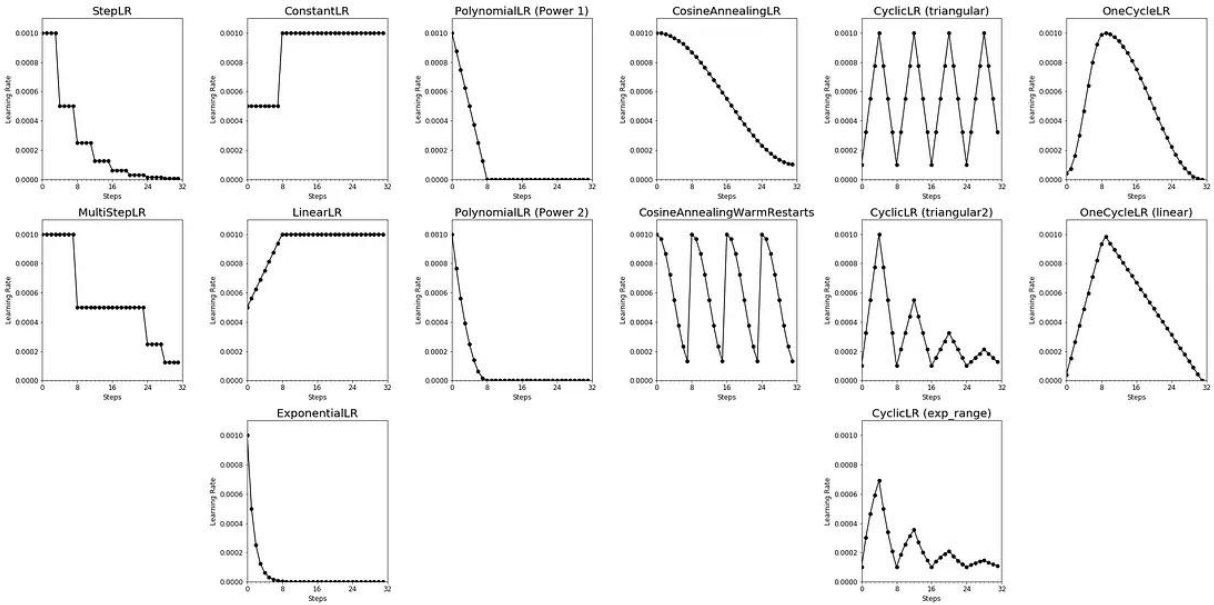


In some cases, two optimization schemes can complement each other, even if both adjust learning rates (eg, adam + StepLR scheduler; latter affects the base learning rate η)

★

What is the intuition behind cosine annealing with warm restarts / warmups?

- By periodically increasing the learning rate and providing a "kickstart" to the optimization process, warm restarts can help the neural network escape local minima and explore different parts of the loss landscape
- Warm restarts reduce the sensitivity to the initial learning rate and learning rate schedule hyperparameters, making it easier to find effective hyperparameters for training.
- it's generally a good idea to include a warmup in the lr schedule (e.g. linearly from 0 to target staring LR), as the gradient distribution without the warmup can be distorted, leading to the optimizer being trapped in a bad local min



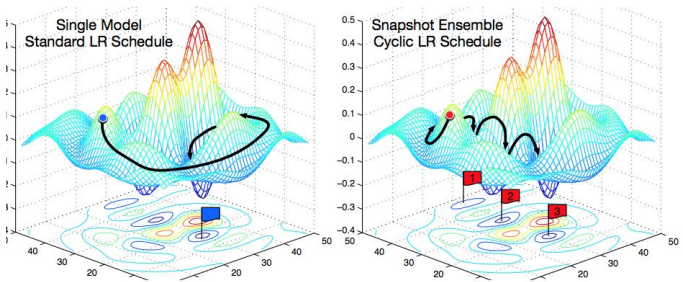
$$\text{Learning Rate} = \eta_{\min} + \frac{1}{2}(\eta_{\max} - \eta_{\min})(1 + \cos(\frac{T_{\text{cur}}}{T_{\max}}\pi))$$

Where:

- $\eta_{\min}$ is the minimum learning rate.
- $\eta_{\max}$ is the maximum learning rate.
- T_{cur} is the current epoch or iteration number.
- $T_{\max}$ is the maximum number of epochs or iterations for one cycle.

In this formula:

- At the beginning of a cycle (when T_{cur} is 0), the learning rate starts at $\eta_{\max}$.
- As T_{cur} increases, the learning rate decreases following a cosine curve to $\eta_{\min}$.
- At the end of the cycle (when $T_{\text{cur}} = T_{\max}$), the learning rate returns to $\eta_{\max}$.



Describe how weights are usually initialized for neural networks. Can you just set them to 0?

- You can't set them to 0 because the model can get "stuck". For example, for ReLU activations, the output will all be 0, and the gradients will be 0. This is a saddle point in parameter space.
- The simplest way to initialize weights is to sample uniformly in a range, such as [-0.2, 0.2], or from a normal distribution such as N(0, 0.1).
- A popular alternative is **Kaiming He normal initialization**, which samples from N(0, sqrt(2/n)) where n is the number of inputs to the node. Intuitively, the 1/n is there to ensure that the input variance is similar to the output variance (recall that if Xs are independent, Var(X+X+...+X) = N*Var(X)).
    - Pytorch actually uses kaiming_uniform initialization by default, which samples uniformly from [-sqrt(6/n), sqrt(6/n)].\
    - Intuitively, **when there is more variability in the inputs (ie, as n gets larger), the range of the distribution shrinks to reduce variability**.

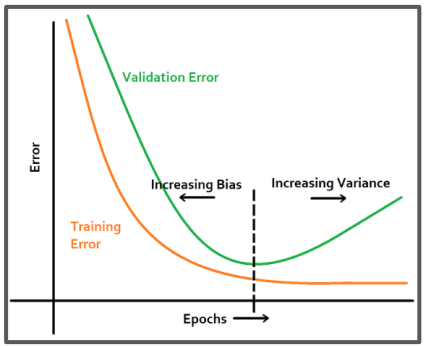
Describe some basic regularization techniques (5).

- **Adding more data**
  - The best (but also, expensive) to avoid overfitting is to add more training data.
  - With enough training data, it can make it so that the models fundamentally has a hard time overfitting on the data, even if it wanted to.
- **L2 Weight Decay**
  - We effectively add a squared term to the cost function (top equation), and when the gradient is taken we are essentially decaying the weight proportional to its size. This limits the expressive power of the neural network.
  - L2 exacerbates the decay on the larger weights; however, L1 is also possible which encourages some features to be 0 completely (discarded).
  - In a way, reduces the parameters of the model without explicitly doing so (ie instead of forcing a lower capacity, the network can learn which nodes to shut off)

$$\widetilde{E}(\mathbf{w}) = E(\mathbf{w}) + \frac{\lambda}{2}\mathbf{w}^2$$

$$w_i \leftarrow w_i - \eta\frac{\partial E}{\partial w_i} - \eta\lambda w_i$$

- **Dropout**
  - When training, only keep a neuron active with some probability p. At test time, all neurons are active.
  - It is necessary to scale outputs at test time appropriately, by multiplying by dropout rate p.
  - Intuitively, this affects the model in a semantic, feature level way and could force it to detect new features for classification, leading to generalization and robustness.
  - Mostly used for the final fully connected layers. It generally is not helpful for convolutional layers.
  - Seems to be falling out of favor, and now mostly batchnorm is used.
- **Data augmentation**
  - Includes horizontal/vertical flipping, random cropping, gaussian noise, rotation, scaling, translation, brightness, contrast, color augmentation.
- **Early Stopping**
  - Tries to stop just before the point where improving the model's fit to the training data comes at the expense of increased generalization error.
  - Can be early stopped when validation performance starts to decrease for a given number of epochs



What is batchnorm and why does it work? Does it go before/after ReLU? What happens at test time? What are some other considerations?

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Implementation Notes:

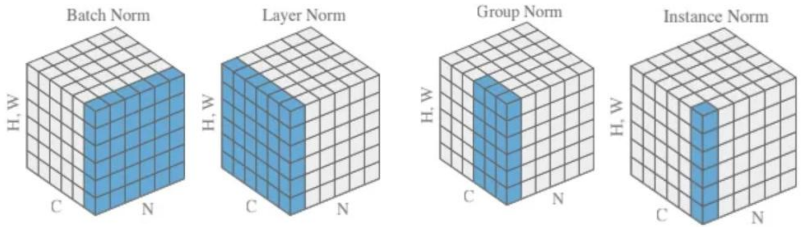
- Makes training **faster and more stable**
- **For convolutional layers**: we want different elements of the same conv filter to be normalized in the same way. Thus, the normalization is performed at the per-channel level, over the batch.
- Batch normalization adds two trainable parameters γ and β to each layer. These ensure that the **expressiveness of the neural network is still constant**.
- **Batchnorm goes before the activations**, generally
    - Rationale is that you will have your data happily centered around the non-linearity, so you get the most benefit out of it. Imagine your data being all <0, then ReLu will have no effect at all; you normalize it: problem solved.
- Usually, when you use batchnorm **dropout becomes unnecessary** (according to one paper, they have the same goals, and when combined lead to worse results).
- It's important to make sure that batch sizes are large enough when using batch norm.
- **At test time**, we would want the output to depend only on the input, deterministically -- not on some "minibatch". Thus, the means and variances used for normalize in this case are from the entire training set, not just at the minibatch level.

Several theories on why batchnorm works:

- **Internal Covariate Shift** for a non-batch normalized network slows down training; by forcing distribution of activations to be uniform, there is less adapting necessary.
    - Inputs to each layer are affected by the parameters of all preceding layers – so that small changes to the network parameters amplify as the network becomes deeper. The change in the distributions of layers' inputs presents a problem because the layers need to continuously adapt to the new distribution. When the input distribution to a learning system changes, it is said to experience covariate shift, occuring at the layer level.
    - For training any given layer, it would be beneficial for the input distribution to remain fixed over time, so that the layers' parameters don't need to readjust to compensate for changes in the input distribution.
- Alternative explanation: Batchnorm instead **smoothes the optimization landscape**, allowing a bigger range of hyperparameters (such as the parameter initialization and the learning rate) to work well. In general, networks that use Batch Normalization are significantly more robust to bad initialization
- Performs a small form of **regularization**, by introducing **stochasticity** (there is normalization per randomized minibatch). Thus, sometimes other techniques like dropout

Explain/compare layer norm, group norm, and instance norm

- Some issues of BatchNorm:
  - BatchNorm fails when the minibatch size is small
  - Harder to parallelize, since there is dependence between batch elements.
  - This is a larger problem for NLP Transformers, not so much for vision currently
  - Normalization constant would be different depending on the length of the sequence
    - So, it is used often for RNNs and NLP in general
- Instead, these norm approaches calculates the statistics per-minibatch sample, instead of per-channel across all minibatches
- Instance norm seems niche and useful to speed up certain generative tasks
- Intuitively, group norm may be able to exploit/utilize some internal/inherent structures within features




```
>>> input = torch.randn(20, 6, 10, 10)
>>> # Separate 6 channels into 3 groups
>>> m = nn.GroupNorm(3, 6)
>>> # Separate 6 channels into 6 groups (equivalent with InstanceNorm)
>>> m = nn.GroupNorm(6, 6)
>>> # Put all 6 channels into a single group (equivalent with LayerNorm)
>>> m = nn.GroupNorm(1, 6)
>>> # Activating the module
>>> output = m(input)
```

How does backpropagation work in neural networks?

Overview

- **Backpropagation** is an automatic differentiation algorithm used to efficiently compute the gradient of the loss function with respect to the weights for a single input-output example (minibatch), to be used for gradient descent. This is necessary because a closed-form, calculus based solution would be intractable.
- Backpropagation's power comes from an important use of intermediate variables within some dependency graph; it can compute gradients in the same time complexity as the forward pass.
- It is a very local process, which makes things elegant & simplified

Forward Pass

$$L = f_k(f_{k-1}(\cdots f_1(x;\theta_1)\cdots;\theta_{k-1}),y)$$

Backwards Pass

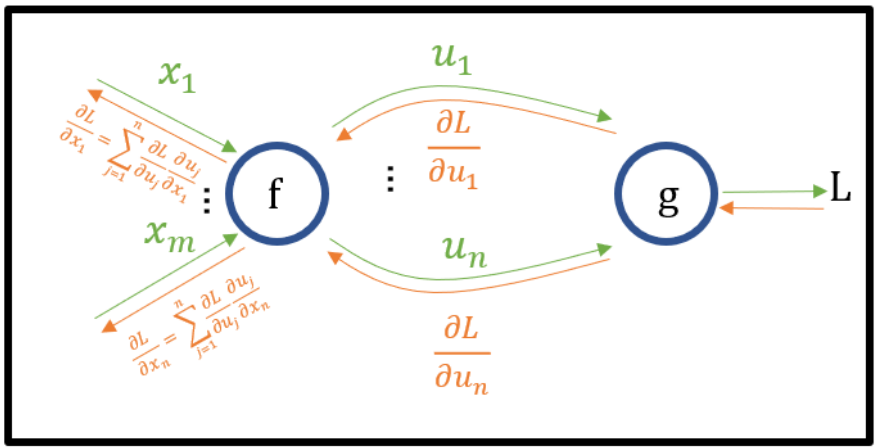
Details

Consider the functional representation for a neural network (left). using the chain rule repeatedly for each layer will yield the partial derivatives w.r.t. each parameter. This is shown below.

Consider any neural network layer $f_i(x) = u$, where $f_i: \mathbb{R}^m \to \mathbb{R}^n$.

The layers after it are represented by $g(u) = f_k(f_{k-1}(\cdots f_{i+1}(u;\theta_{i+1})\cdots;\theta_{k-1});\theta_k) = L$, where $g: \mathbb{R}^n \to \mathbb{R}$.

As shown below, the input to the layer is $x \in \mathbb{R}^m$, the layer's output is $u \in \mathbb{R}^n$, and the output is fed to rest of the network g to produce the loss value L.



"Upstream gradient" (based on later layers)

"Local gradient" (based on Jacobian)

$$\frac{\partial L}{\partial x_i} = \sum_{j=1}^{n} \frac{\partial L}{\partial u_j}\frac{\partial u_j}{\partial x_j}$$

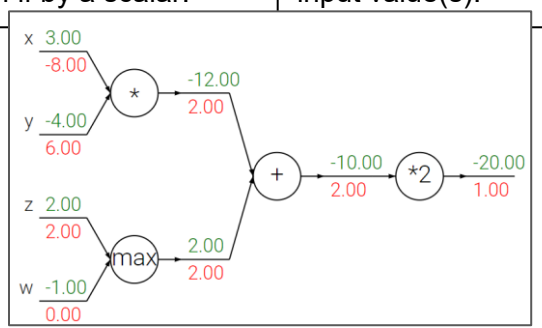
Impact of x_i on L wrt all outputs u_j are added to get overall impact

Notes:

- In general, a differentiable function can be part of a computational graph for backpropagation if there are two functions properly defined:
  - Forward pass (with some values cached for efficiency)
  - Backward pass (which takes in the upstream gradient, and computes gradients for the inputs using the local gradient and chain rule)

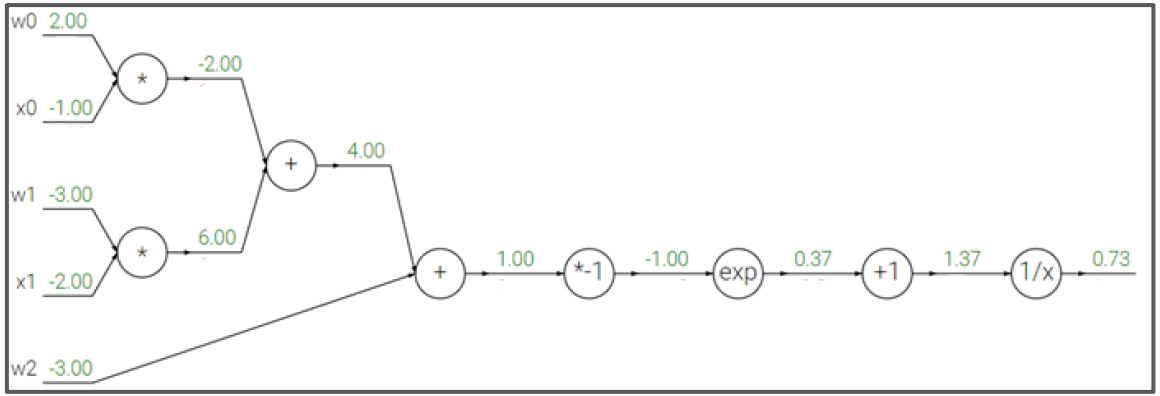
Describe how the following functions/gates propagate the upstream gradient during backpropagation: add, max, multiply

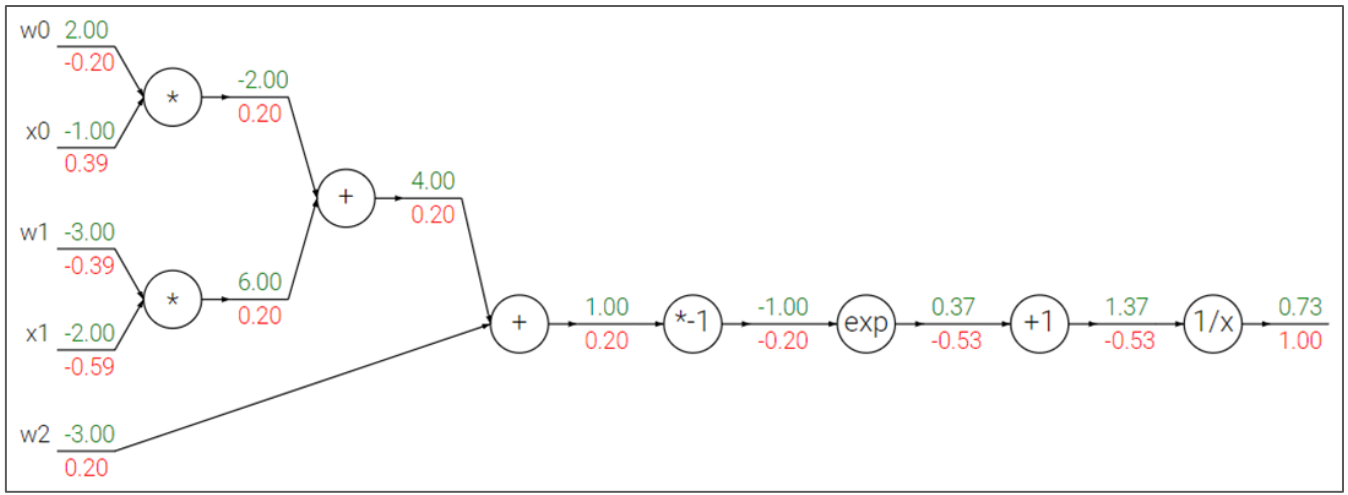
Gate	Behavior	Intuition
Add	Adds all the upstream gradient on its output and distributes that sum equally to all of its inputs. This follows from the fact that the local gradient for the add operation (e.g. x+7) is simply +1.0	Forwards gradients, unchanged.
Max	Distributes the gradient (unchanged) to exactly one of its inputs (the input that had the highest value during the forward pass). This is because the local gradient for a max gate is 1.0 for the highest value, and 0.0 for all other values. (Since this isn't differentiable when there are ties, ie max(x,y) when x=y, softmax is typically used instead)	Routes the gradient to the largest forward pass input value.
Multiply	Local gradients are the input values (except switched), and this is multiplied by the gradient on its output during the chain rule. This is regular multiplication if by a scalar.	Multiplies gradient by the other input value(s).



Complete the following computational graph for backpropagation.

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$

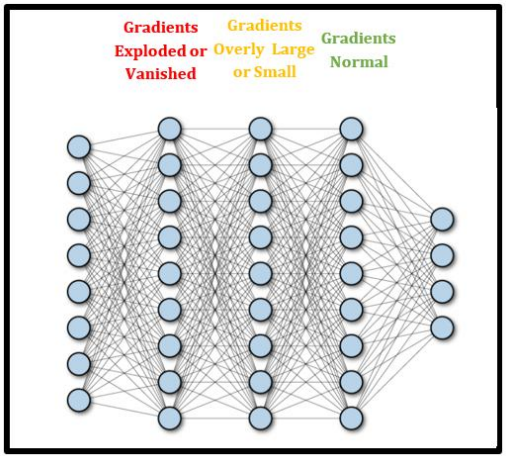




What is the “straight-through estimator” and when is it used?

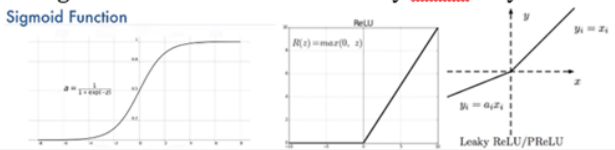
- Method used to approximate gradient during backwards pass when dealing with operations in the forwards-pass that are non-differentiable or whose gradients are difficult to compute
- For the purposes of the backwards pass, replaces the function by the identity function when computing the gradients with respect to the input
  - Assume/approximates that the function is not doing anything special, and just propagates the gradient further upstream
- Example usages:
  - **Quantization Aware Training**: quantization is non-differentiable, but we just treat the operation with gradient 1
  - **VQ-VAE**: Finding nearest neighbor codebook involves an argmin operation. Straight-through estimator directly passes gradients back to the selected codebook entry

What is the vanishing/exploding gradients problem?



- Recall that in NN optimization, after the forward pass, a backwards pass (by backpropagation) is taken to obtain the partial derivative of the loss with respect to each parameter. Then, a gradient descent step is taken for each parameter to update its weights.
- Unfortunately, the scales of these gradients can be an issue. They can be too small, leading to near-zero step (vanishing gradients). Or they can be too large, leading to huge steps (exploding gradients).
- Some general solutions that works for both problems:
  - Standardizing/normalizing data
  - L2 regularization
  - Proper weight initialization
  - BatchNorm

	Vanishing Gradients	Exploding Gradients
Underlying Issue	Due to the chain rule's multiplicative property, gradients accumulate from the last layer to the first such that they become increasingly excessively small after repeated multiplications with numbers whose magnitude is less than 1. The signal dies out.	Due to the chain rule's multiplicative property, gradients accumulate from the last layer to the first such that they become increasingly excessively large after repeated multiplications with numbers whose magnitude is larger than 1. The signal becomes amplified to an impractical level.
Signs/Symptoms	• Loss changes very slowly • Weights near the output layer change much more than those near the input layer	• Loss becomes NaN • Large swings in loss function • Poor loss performance • Parameter weights become huge or NaN

Solutions	• Use non-saturating activation functions
	  ○ For example, sigmoid (and ReLU to some extent) yields local gradients close to zero. So Leaky ReLU may be better.
	 
		• Lowering learning rate
		• Gradient clipping, so that the step size never exceeds a threshold.
		  ○ This can either by capping each derivative value with a max/min, or scaling the entire norm (all gradients together, as if they were concatenated) to a max if it exceeds it.

- Residual connections

What are some general tips/tricks on tuning these hyperparameters? Learning rate, batch size.

Learning Rate:

- Perhaps the most important hyperparameter. Gridsearch on a validation set usually suffices.
- **Learning rate warmup** can be useful way to reduce the effects of “early overfitting” to the first training samples, and reducing risk of starting with a bad descent direction
- A potentially complementary approach is **reducing learning rate at end**; at that time, you are close to the optimum, so you need to be careful about the step size. This may be less of a problem earlier, as a larger step in the gradient direction will lead to gains

Batch Size:

- While a larger batch size will lead to a more accurate estimation for the gradient, there’s a lot of evidence that smaller (and less exact) batch sizes work better
- Large-batch methods tend to converge to sharp minimizers of the training function
- The noise from small batch sizes can actually help an algorithm jump out of a bad local minimum and have more chance of finding either a better local minimum
- However, there are some tricks/heuristics that seem to allow larger batch sizes:
    - Slowly scaling up the batch size and learning rate together **(larger latch size -> more accurate estimation of gradient -> allows for higher LR)**
    - Initializing batchnorm params as 0

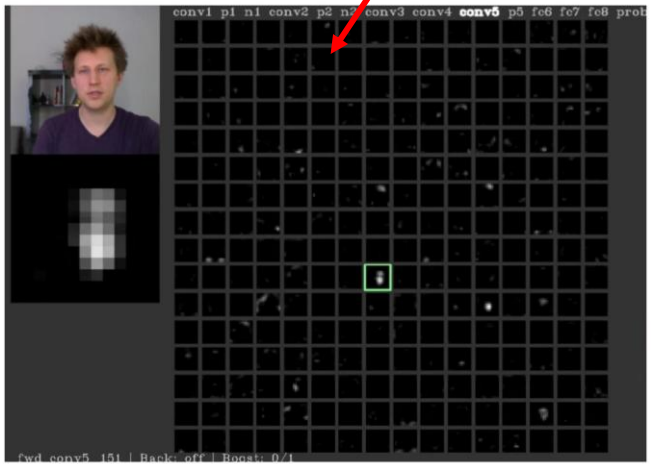
What are some sources of randomness/stochasticity in neural networks? Why is this a good thing?

- Randomly **initializing weights/biases**
  - Useful for **ensembles** (e.g. random forests), reproducibility experiments
- Regularization techniques like **dropout**, **data augmentation**
  - in the limit, the randomness allows exposure to all the possible data augmentation/dropout configurations
  - It forces the network to learning meaning representations instead of memorizing
- Minibatch sampling like **SGD**
  - Injects some **variability in the gradient steps** to improve optimization to do some “searching” and “bouncing” out of local minima
- Batchnorm also involves randomness, and creates a regularization effect

What's a dead neuron, how can they be detected, and how can they be prevented?

- This means that for a given FC layer neuron or feature map, its weights are such that it always outputs values very close to zero after the activation function, regardless of input
  - If the activation function is ReLU, this implies the logits are negative, either due to the weights or a large negative bias term
- This can be caused by a learning rate that's too high
- Once a ReLU ends up in this state, it is unlikely to recover, because the function gradient at 0 is also 0, so gradient descent learning will not alter the weights
  - "Leaky" ReLUs with a small positive gradient for negative inputs (y=0.01x when x < 0 say) are one attempt to address this issue and give a chance to recover
  - Sigmoid and tanh neurons can suffer from similar problems as their values saturate, but there is always at least a small gradient allowing them to recover in the long term.

Possibly dead activation map if empty for many data inputs



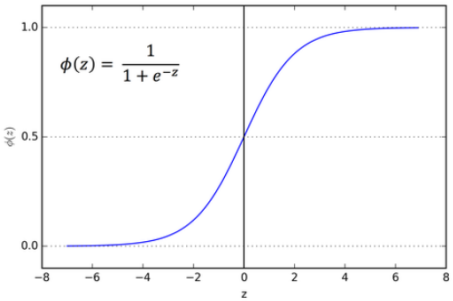


Fig: Sigmoid Function

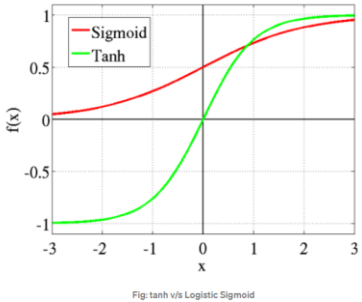
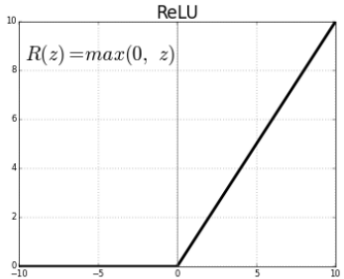
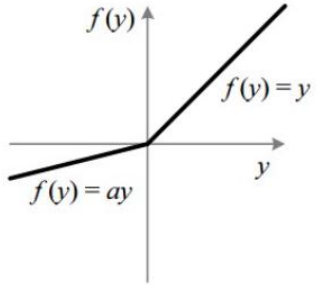


Fig: tanh v/s Logistic Sigmoid

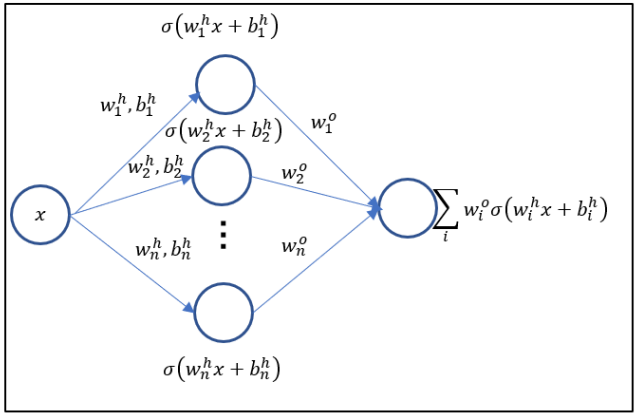


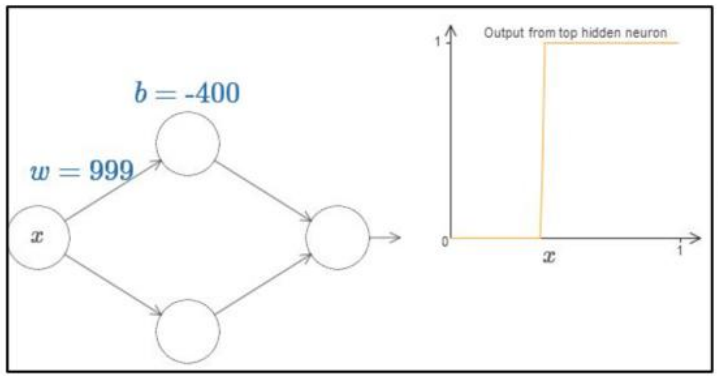


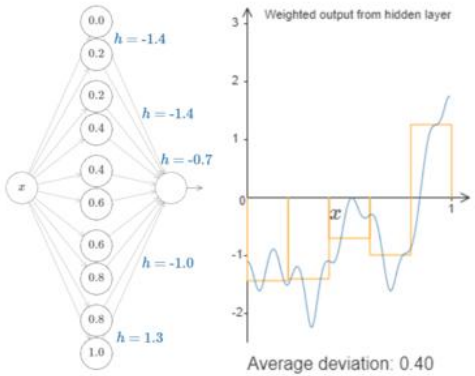
Leaky ReLU (usually a is around 0.01)

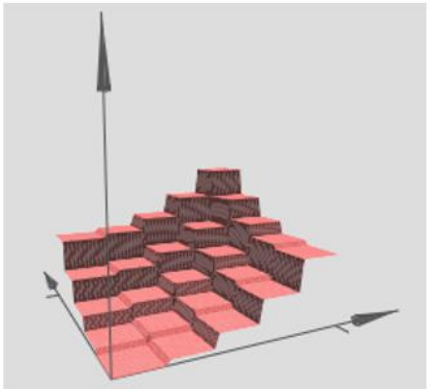
What is the universal approximation theorem, and what are its limitations for practical applications?

- A result that states that a FC network with only one hidden layer can approximate any function to arbitrary precision, given enough width (hidden neurons)
- At a high level, this is because sigmoid can represent step functions, and when enough are combined, can approximate any function
- In practice, there is overwhelming empirical evidence that deep conv networks are more generalizable and accurate, due to architectural priors, data limitations, and computational limitations.
    - But still an important finding that suggests neural networks are asymptotically unbiased in principle



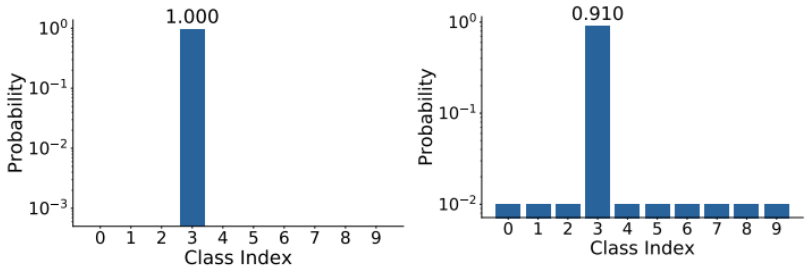






What is label smoothing?

- **Label smoothing**
  - Intuitively, tries to reduce overconfident predictions and improve **calibration**
  - Accounts for the fact that datasets may have mistakes in them

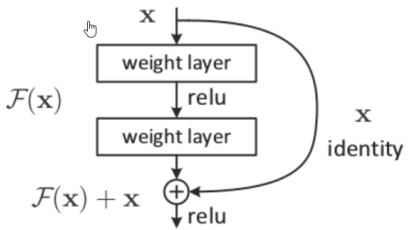


(a) Hard Label (b) LS

Seminal & Foundational Topics in Deep Learning

What are the benefits of residual connections in neural networks?

- Alleviates the vanishing gradients problem by allowing gradients to flow directly through the skip connections, maintaining a stronger gradient for longer
    - This in turn allows for deeper networks, improved training speed, and better convergence
- Allows learning identify functions easier
- Simplifies the learning problem, since each residual block only needs to learn the residual part (difference) with respect to the identity function

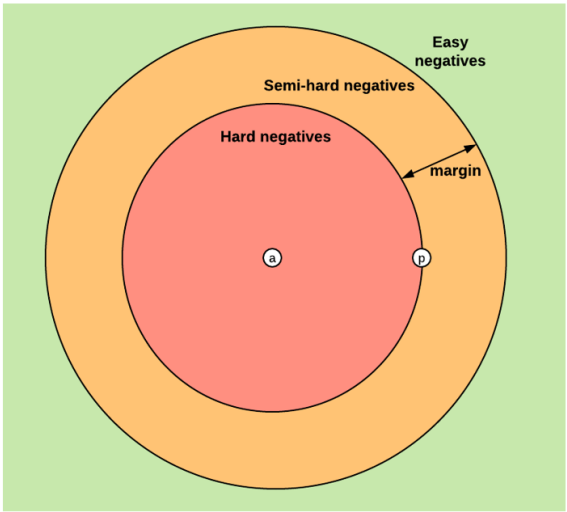


What is the triplet loss, and when is it used? How can it be trained effectively?

- **Embeddings for classification** are useful in cases where we have a variable number of classes (not fixed), for example face verification.
- **Embeddings for retrieval** is a natural fit, where you embed your test query and use k-nn in the embedding space to retrieve the top k entries.
- The **triplet loss** provides a way to learn good embeddings
  - Intuitively, any two examples with the same label should be close in the embedding space, and any two examples with different labels should be far away

$$\mathcal{L}(A, P, N) = \max\left(\|\mathrm{f}(A) - \mathrm{f}(P)\|^2 - \|\mathrm{f}(A) - \mathrm{f}(N)\|^2 + \alpha, 0\right)$$

  - A is the anchor
  - P is an example with the same class as the anchor (“positive”)
  - N is an example with a different class compared to anchor (“negative”)
  - f is an embedding function
  - α is the margin between positive and negative examples.
  - Higher margins mean better embeddings (generally), but will also make training harder
- There are N^3 possible triplets; for efficiency, we want to select good triplets to learn from.
- In FaceNet, they use random **semi-hard negatives**, recomputing after each epoch. There are 3 categories, given a fixed anchor and positive and relative to the negative:
  - **easy triplets**: triplets which have a loss of 0, because $d(a,p) + margin < d(a,n)$
  - **hard triplets**: triplets where the negative is closer to the anchor than the positive, i.e. $d(a,n) < d(a,p)$
  - **semi-hard triplets**: triplets where the negative is not closer to the anchor than the positive, but which still have positive loss: $d(a,p) < d(a,n) < d(a,p) + margin$



What is LoRA?

- LoRA (Low Rank Adaptation) is a method for efficient fine-tuning of LLMs by injecting low-rank trainable matrices into the model architecture, while keeping other components frozen

Instead of updating the full weight matrices in fine-tuning, LoRA proposes to:

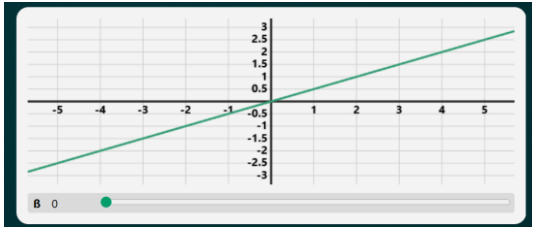
- **Freeze the pre-trained weights** W_0.
- **Inject low-rank decomposition matrices** $\Delta W = BA$, where $B \in \mathbb{R}^{d \times r}$, $A \in \mathbb{R}^{r \times k}$, and $r \ll \min(d, k)$.
- Modify the forward pass from Wx to $W_0x + BAx$.

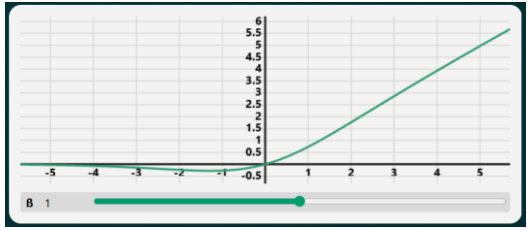
This reparameterization allows training only the small matrices A and B, while keeping the backbone frozen.

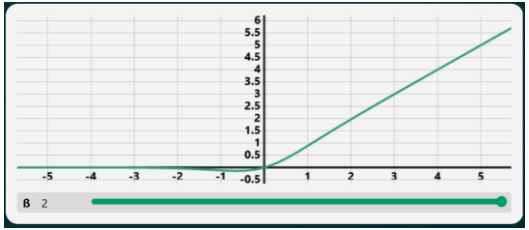
- This is applied to self-attention layers: query & value projection matrices W_q, W_v
- Recall that the rank of a matrix roughly tells you how much unique information it contains & its egress of freedom. Note that LLMs are already heavily over-parameterized and contain general knowledge
  - Adapting them often just means highlighting or amplifying some specific features
  - These changes often live in a low-rank subspace of the full weight space

What is the SwiGLU activation?

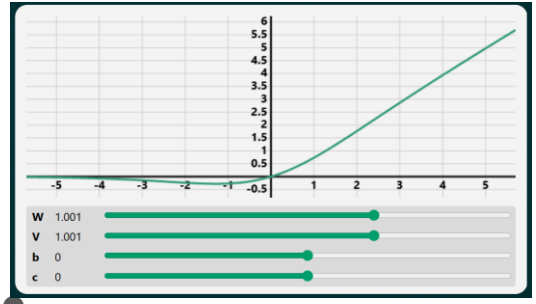
- **Swish Activation**: $swish(x) = x * sigmoid(\beta x)$
  - β is a learnable parameter; when $\beta = 0$, becomes linear; when $\beta \to \infty$, becomes ReLU

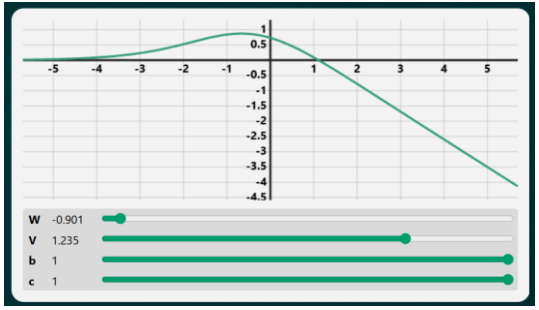


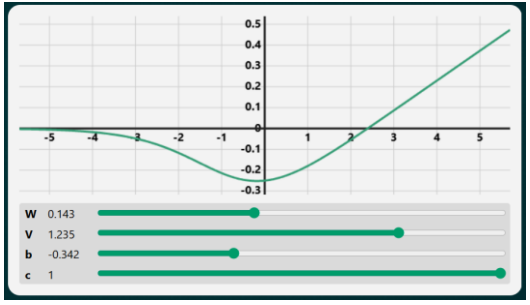




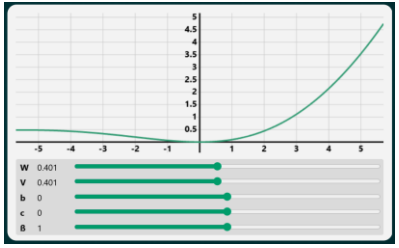
- **GLU (Gated Linear Unit) Activation**: $GLU(x) = (Wx + b) * sigmoid(Vx + c)$
  - W, V, b, c all learnable parameters
  - Can be interpreted as a linear transformation over x, times the probability of activating that neuron

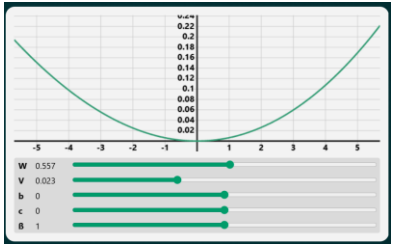


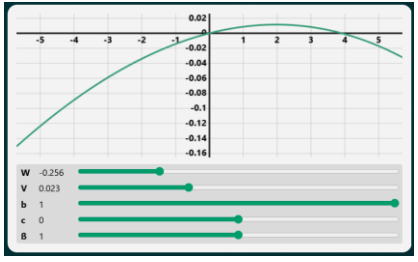


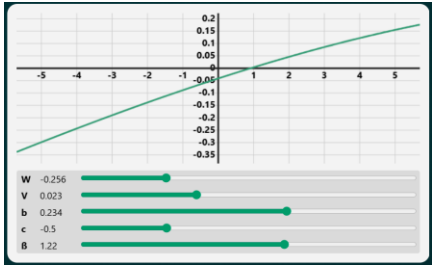


- **SwiGLU**: Swi$GLU(x) = (Wx + b) * swish(Vx + c)$
  - W, V, b, c all learnable parameters
  - No real concrete interpretation, except that it's even more expressive









Explain how ConvNeXt V2 works

- SOTA CNN (no transformers)
- Uses masked autoencoders for self-supervised learning
  - Masks image regions and tries to reconstruct the missing regions.
  - Uses **sparse convolutions** for efficiency since there are many zeros
  - loss function is computed only on the masked parts of the image
- Global Response Normalization layer
  - Address issue of feature collapse during self-supervised learning
  - designed to enhance feature diversity and promote competition across feature channels, which leads to better representation learning and overall performance

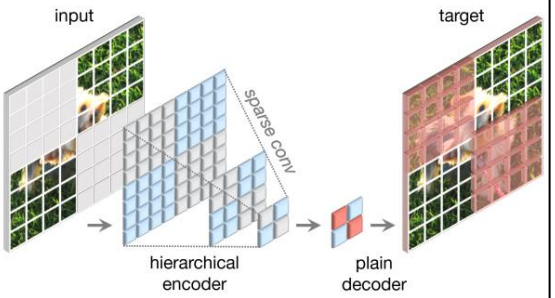


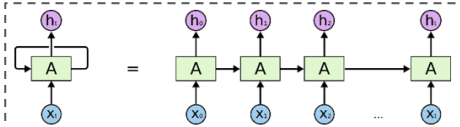
Figure 2. **Our FCMAE framework.** We introduce a fully convolutional masked autoencoder (FCMAE). It consists of a sparse convolution-based ConvNeXt encoder and a lightweight ConvNeXt block decoder. Overall, the architecture of our autoencoder is asymmetric. The encoder processes only the visible pixels, and the decoder reconstructs the image using the encoded pixels and mask tokens. The loss is calculated only on the masked region.

Neural Networks Designed for Sequential Data

(RNNs, LSTMs, Transformers)

What can RNNs be used for, and how do they work?

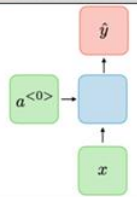
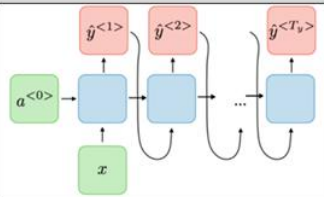
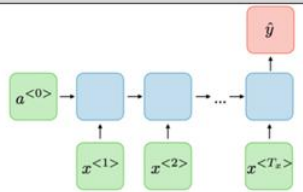
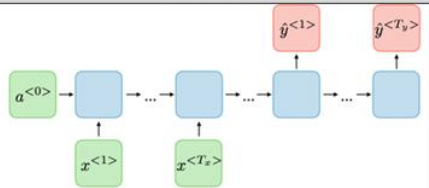
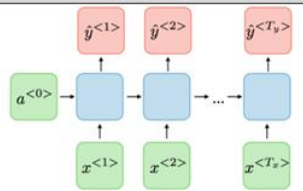
- **Recurrent Neural Networks (RNNs)** allow you to work with nonfixed, variable-length sequential data, e.g. time series data or sentences, by building a hidden state over time.
- During training, we do **backpropagation through time**, for some number of timesteps that we choose. However, **vanishing/exploding gradients** is a significant problem.
- In practice, **long term dependencies** are an issue; it's difficult to "remember" previous context when the time step gap is very large, since repeatedly overwritihg a hidden state leads to data loss.
- In ordinary RNNs, there is no **bidirectionality**; for many-to-one or many-to-many settings, we can only use the inputs which we have already seen when making a prediction, and can't use later input elements.
    - However, there are ways to address this, e.g. Bidirectional RNNs which read left-to-right and right-to-left in parallel.
- RNNs and its variants are related to **Bayesian filtering**. Both work with sequential data/measurements to update an internal state, to predict a target variable.
    - Bayesian filters provide probabilities and are more "specialist", useful in settings where the noise and dynamics are well-characterized
    - RNNs tend to be more general and are universal approximators, but require much more data and computing resources



Composed of an internal hidden state which updates itself with each input x.

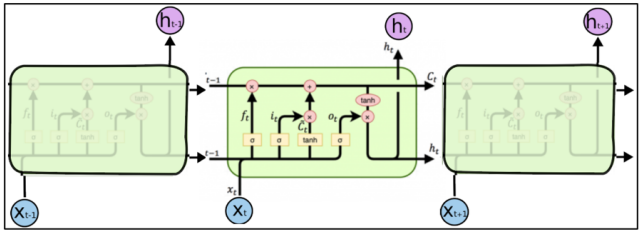
At some time steps, we will ask it to predict a vector.

- In general, $h_t = f_w(h_{t-1}, x_t) = hidden\ state\ at\ time\ t$
    - $f_w = some\ function\ with\ parameters\ w$
        - Weights and function are fixed – don't change with time
    - $h_{t-1} = previous\ time\ step's\ hidden\ state$
    - $x_t = input\ @\ time\ t$
- In general, $y_t = f_y(h_t) = output\ for\ time\ t$
- A "vanilla" version uses matrices & tanh nonlinearities for f_w, f_y
    - $h_t = tanh\ (W_{hh}h_{t-1} + W_{xh}x_t + b_h)$
    - $y_t = W_{hy}h_t + b_y$

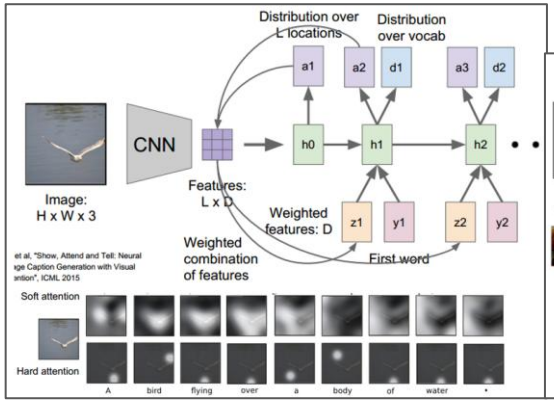
I/O Setup	One to One (vanilla feedforward network)	One to Many	Many to One	Many to Many (indirect)	Many to Many (direct)
Graph					
Remarks	A vanilla feedforward NN.	Previous output prediction fed back in as input ("**autoregression**"). Can be used as a sequential decoder.	Can be used as a sequential encoder.		
Example Applications	Image classification, monocular depth est, SVR	Image captioning, music generation from text.	Sentence sentiment classification, video classification.	Language translation, visual question answering (VQA; image+sentence input, sentence output)	Temporally smoothed video tracking or depth estimation

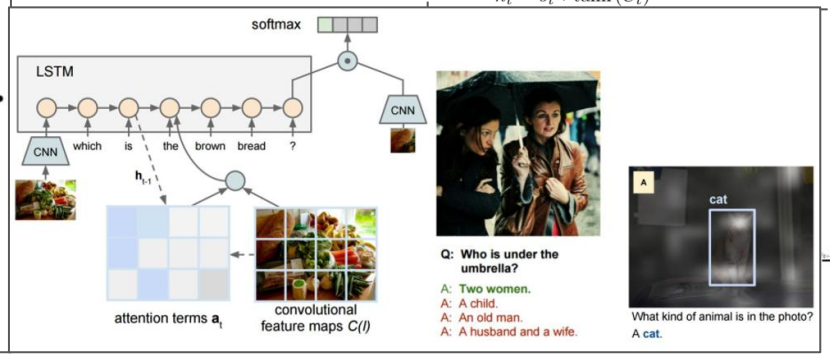
What are LSTMs used for and how do they work?

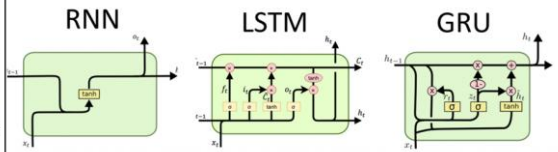
- **Long Short Term Memory (LSTM)** is a RNN variant solves issues with exploding/vanishing gradients by two modifications, which improve gradient flow (at a high level, similar to ResNets' residuals):
  - Incorporate a **cell state** (c) used only internally for each time step. The hidden state (h) is still used for the output.
  - Replacing the simple tanh/matrix multiplication update rule with a **gating mechanism** which updates both the cell and hidden states.
  - The gating mechanism determines what to forget, update, & reveal in the output (vs keep in cell state)
  - Allows for discarding irrelevant information
- There are many variants on LSTMs with different configurations, e.g. adding/removing gates, or only using hidden states (no cell state).
  - A popular example is the **Gated Recurrent Unit (GRU)**
  - Some research finds that GRUs do better, some say that they do the same. The consensus seems to be that you should try both
- They have been used for many tasks involving NLP, such as:
  - **Image captioning with attention**
  - **Visual question answering (VQA)**



- An LSTM module has a hidden state (h, will be used for output) and a cell state (c, only used internally) for each time step
- There are three gates:
  - **f (forget gate):** Decides what information from the cell state should be thrown away or kept, using a sigmoid in [0,1].
    - $f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$
  - **i (input/information gate):** Decides which values should be updated in the cell state, using a sigmoid. Additionally, the update magnitude and direction is determined by a tanh.
    - $i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$
    - $\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$
  - **o (output gate):** Decides how much to reveal/filter the cell state to the hidden state, to be output. This is done with a sigmoid.
    - $o_t = \sigma\left(W_o \; [h_{t-1}, x_t] \; + \; b_o\right)$
- Overall, the cell state is updated with the forget gate and input gate outputs:
  $C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$
- The hidden state is updated with the current cell state and the output gate's filter:
  $h_t = o_t * \tanh\left(C_t\right)$

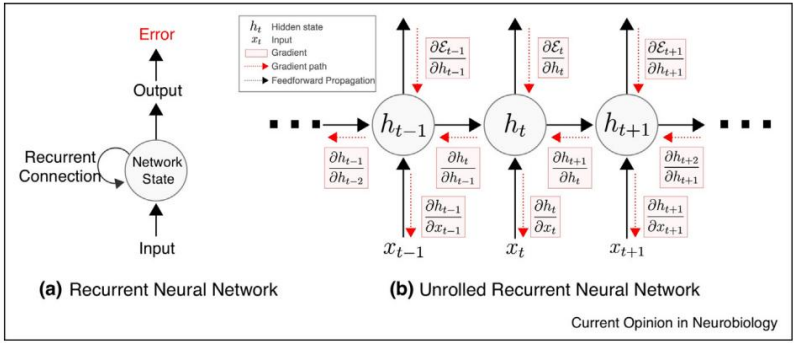


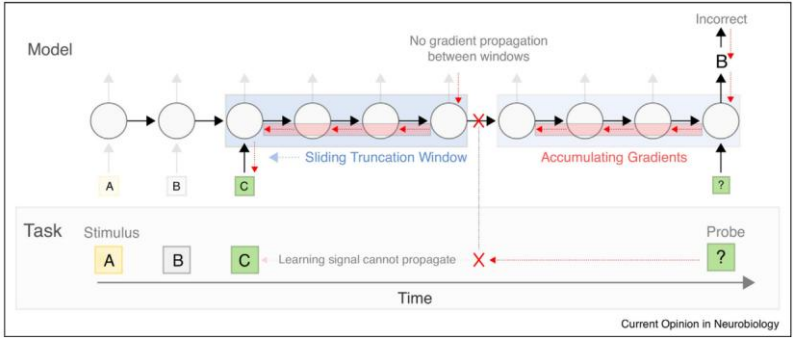




How does backpropogation through time (BPTT) work?

- Enables learning of temporal dependencies by computing gradients across all timesteps of an RNN forward pass
  - In contrast, regular backpropagation only cares about the current input-output mapping, not how past inputs influence the present.
  - Intuition: Learning a musical instrument. If you hit the wrong note in bar 8, the mistake could be because of something you did in bar 5 (e.g., bad hand positioning). You need to trace the error *back through time* to understand and correct it.
- Challenges:
  - Vanishing/exploding gradients as they pass through time steps
  - Computationally expensive
- **Truncated BPTT** limits the number of time steps we backprop through for efficiency & stability
  - Intuitively, we assume that the influence of things that happened more than N steps ago is small enough that we can ignore it for training purposes
  - Resulting model might fail to capture long-term dependencies
  - Previous hidden states still influence future ones – we just don't train the network to fully learn how changes in earlier states affect later ones





$$\frac{\partial E}{\partial h_t} = \frac{\partial E}{\partial h_T}\frac{\partial h_T}{\partial h_t} = \frac{\partial E}{\partial h_T}\prod_{k=t}^{T-1}\frac{\partial h_{k+1}}{\partial h_k}$$

In NLP, what are Transformers and how do they work at a high level? How do they differ from LSTMs/RNNs?

There are several significant differences between vanilla Transformers and RNNs:

- They use an encoder-decoder NN structure, and encoder inputs are **fed all at once**; recurrent hidden states are completely removed
  - This allows for very **explicit modeling of long-range dependencies** compared to RNNs/LSTMs
  - Workarounds like **"chunking"** are necessary for inputs larger than the maximum size
  - Does not need to process the beginning of the sentence before the end, allowing for **parallelism** during training
  - Along with the use of skip connections, this partially **addresses the BPTT vanishing/exploding gradients**
  - To encode order/time series information, **positional encoding** is added to the inputs
- Adopts a **self-attention mechanism**, which differently weighs the significance of each part of the input data, dynamically on-the-fly during inference.
  - Compared to normal learned weights, these can be considered "**dynamic data-dependent weights**" or "fast weights"
  - Intuitively, this enables "selective" **memory** to attend to specific relevant parts of the input
- Two well-known models based on Transformers are:
  - **BERT** is encoder-only and is non-autoregressive. It can be thought of as an embedding function which, with extra linear layers, can be used for downstream tasks
  - **GPT-3**, decoder-only and is autoregressive. Trained in a self-supervised way by next-word prediction, and can be easily fine-tuned for specific tasks.

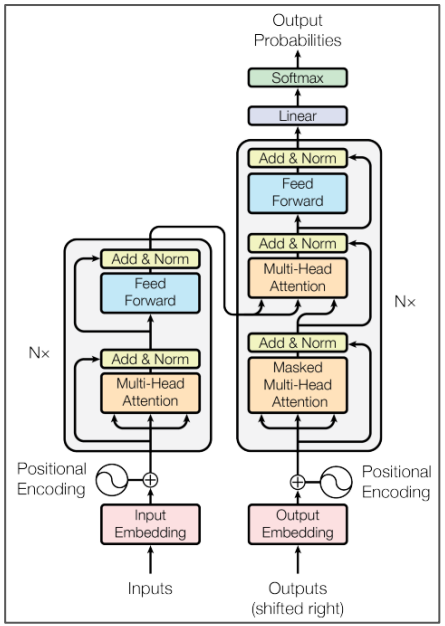
Encoder input preparation:

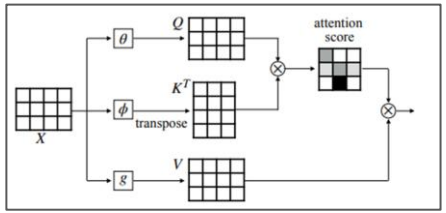
- Input words tokenized as fixed length vectors (e.g. 512 dims)
- Positional encodings are added to each word embedding
- With inputs in matrix X, this is fed into the first encoder blocks
- Output is a contexualized embedding, per-token that is richer than the initial word embeddings since it looks at the surrounding context

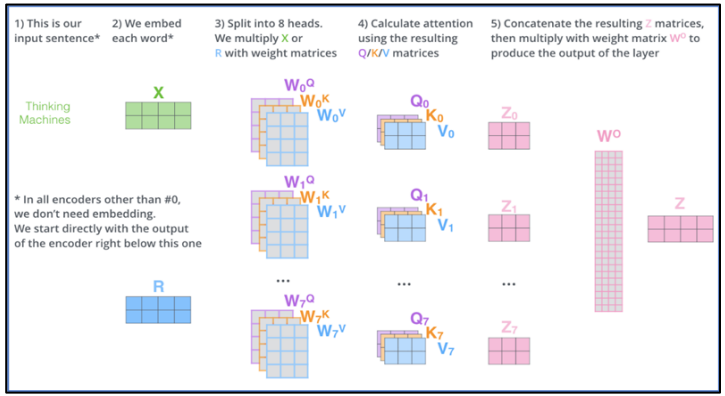
Decoder is similar to the encoder, with a few changes:

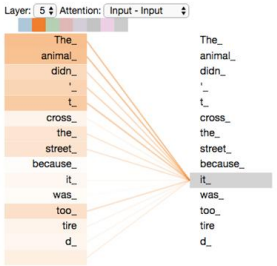
- Decoder performs **autoregression** repeatedly predicts the next word (with a linear layer which is the size of the corpus and softmax, at top), refeeding that prediction back into itself to obtain the next input
- **Masked self-attention layer** is only allowed to attend to earlier positions in the output sequence
- **Multi-Head Attention layer** works just like the encoder, except it creates its Queries matrix from the layer below it, and takes the Keys and Values matrix from the last encoder stack

- Useful analogy: Attention is an aggregation operation that allows tokens to communicate with each other (reduce); MLPs process each token individually (map). So it's a bunch of map-reduce operations.









Given a text sequence with L_t tokens each of a d_t dimensional embedding, and an image patch sequence with L_i tokens each of a d_i dimensional embedding, write the equations associated with text attending to image (# heads=h).

- $X_t \in \mathbb{R}^{L_t \times d_t}$: Text sequence with L_t tokens, each of a d_t dimensional embedding
- $X_i \in \mathbb{R}^{L_i \times d_i}$: Image patch sequence with L_i tokens, each of a d_i dimensional embedding
- $W_{Q,j} \in \mathbb{R}^{d_t \times d_k}$
- $W_{K,j} \in \mathbb{R}^{d_i \times d_k}$
- $W_{V,j} \in \mathbb{R}^{d_i \times d_v}$
- $Q_j = X_t W_{Q,j} \in \mathbb{R}^{L_t \times d_k}$; intuitively, what you're currently focusing on or asking about (here, text)
- $K_j = X_i W_{K,j} \in \mathbb{R}^{L_i \times d_k}$; intuitively, the tags/label of what you're referencing (here, image patches)
- $V_j = X_i W_{V,j} \in \mathbb{R}^{L_i \times d_v}$; intuitively, the content / actual information of what you're referencing (here, image patches)
- $A_j = softmax\left(\frac{Q_j K_j^T}{\sqrt{d_k}}\right) \in \mathbb{R}^{L_t \times L_i}$; intuitively, what part of the image each word should focus on. Softmax performed per-row.
- $h_j = A_j V_j \in \mathbb{R}^{L_t \times d_v}$; intuitively, gathering information & visual context relevant to each word
- $W_O \in \mathbb{R}^{h d_v \times d_t}$
- $O = concat(h_1, \ldots, h_h) W_O \in \mathbb{R}^{L_t \times d_t}$
- (often, $d_t = d_i = d$ which then reduces to the same setup as self-attention)

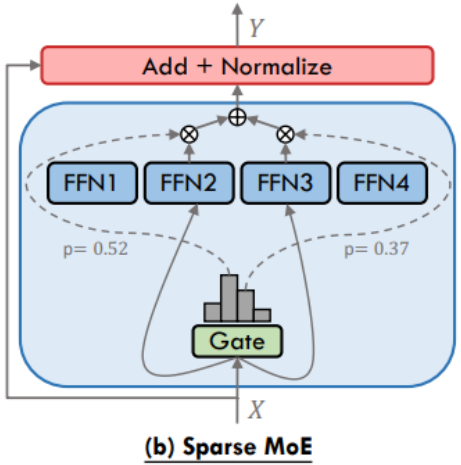
What's the difference between models like BERT & GPT?

Feature	**GPT**	**BERT**
Name	*Generative Pre-trained Transformer*	*Bidirectional Encoder Representations from Transformers*
Architecture	Decoder-only Transformer	Encoder-only Transformer
Pretraining Task	Causal Language Modeling (predict next token using only past context; uses causal masking to ensure it can't cheat by looking at future words)	Masked Language Modeling (predict missing tokens, eg "The [MASK] is blue" → predicts "sky")
Training Direction	Left-to-right (unidirectional)	Bidirectional
Usage	Text generation: Chatbots, text generation, code generation, story writing, etc.	For when you want rich, contextualized embeddings per input token. Used for text understanding/comprehension, sentiment analysis, question answering, classification, named entity recognition, etc. Also used in retrieval-augmented generation to retrieve documents to later pass into encoder.
Examples	GPT-2, GPT-3, GPT-4	BERT, RoBERTa, DistilBERT

In principle generative models like GPT could also use an encoder to achieve better understanding of the input prompt, but is generally avoided for simplicity & scalability, and that in many cases does not lead to a significant gain in performance.

Describe how mixture-of-experts (MoE) models are formulated.

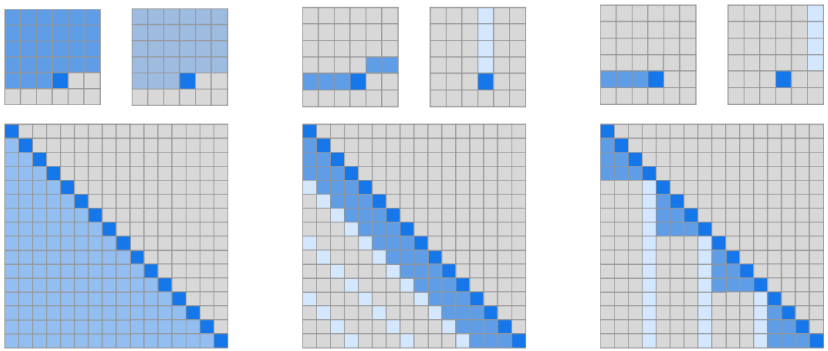
- Note that the vanilla transformer block's FFN is a dense network, since every component is activated on every forward pass; MoE layer provides a sparse model/network
- Router is a FFN that does a n-way classification for N experts. Can choose one expert or multiple and compute weighted mean
- Still need to load all parameters, but speeds up inference since you don't need to go through all weights
- Can lead to tricker training due to complexity and risk of overfitting, collapse, etc
  - Eg can introduce load balancing loss to encourage uniform routing



(b) Sparse MoE

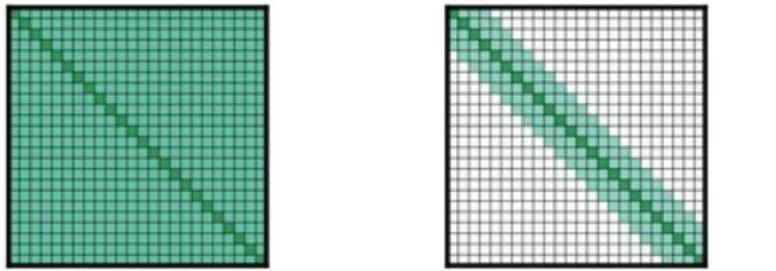
Explain how sparse/windowed attention works.

- Reduces the quadratic complexity of the standard attention mechanism by computing attention over a subset of token pairs rather than all token pairs
  - From $O(dn^2)$ to $O\big(dnlog(n)\big)$ or $O(dn)$, where n is sequence length and d is the hidden size



(a) Transformer (b) Sparse Transformer (strided) (c) Sparse Transformer (fixed)

Figure 3. Two 2d factorized attention schemes we evaluated in comparison to the full attention of a standard Transformer (a). The top row indicates, for an example 6x6 image, which positions two attention heads receive as input when computing a given output. The bottom row shows the connectivity matrix (not to scale) between all such outputs (rows) and inputs (columns). Sparsity in the connectivity matrix can lead to significantly faster computation. In (b) and (c), full connectivity between elements is preserved when the two heads are computed sequentially. We tested whether such factorizations could match in performance the rich connectivity patterns of Figure 2.



(a) Full n^2 attention (b) Sliding window attention

Explain how KV Caching works.

- Unlike during training, with autoregressive inference we only use a single query token (the previously predicted output token)
- As a result, we only need to compute 3 vectors Q_t, K_t, V_t and can use cached values for $K_{1:t-1}, V_{1:t-1}$
- Results in $O(t)$ instead of $O(t)^2$ time per-token

With KV caching:

- **Cache** $K_1, \ldots, K_{t-1}$ and $V_1, \ldots, V_{t-1}$ from previous steps.
- At time t:
  - Compute only Q_t, K_t, V_t
  - Append K_t, V_t to cache:

$$K_{1:t} = \text{concat}(K_{1:t-1}, K_t) \in \mathbb{R}^{t \times d_k}$$
$$V_{1:t} = \text{concat}(V_{1:t-1}, V_t) \in \mathbb{R}^{t \times d_v}$$

- Attention becomes:

$$\text{Attention}_t = \text{softmax}\left(\frac{Q_t K_{1:t}^\top}{\sqrt{d_k}}\right) V_{1:t}$$

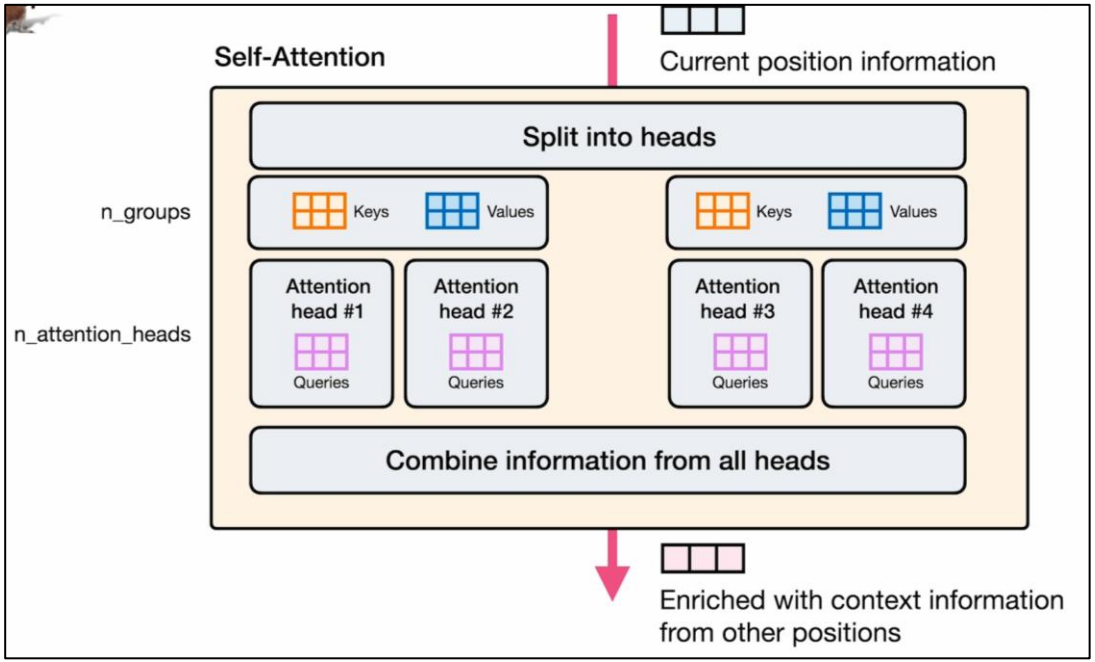
You **never recompute** any K_i, V_i for $i < t$. You just **retrieve** them from cache.

How is training and autoregressive inference different for decoder-only transformer LLMs?

	Training	Inference
Causal Masking	Essential, or else network can cheat in the next token prediction	In principle, not required since there is no “cheating” by construction. However, we still don’t want previously predicted tokens to attend to later predicted tokens after it because it’ll cause a train-test mismatch
Queries	Use sequence of queries. Consequently, predicts next token at all positions	Only use the latest previous generated token as query. Consequently, predicts one token at a time.
KV Caching	Not used, since we are training over randomized batches	Used to speed up inference

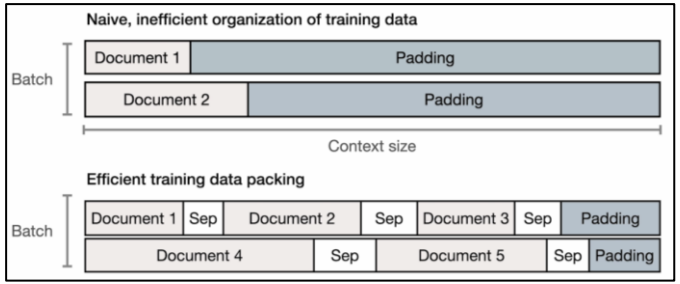
What is Grouped Query Attention?

- Variant of multi-head attention, where several groups share the same keys and values; still one query per head.
- Reduces memory and latency, especially with KV-caching



How does RoPE work?

- Relative positional encoding is generally better than absolute
  - Lets model generalize better across sequence lengths
  - Learns “distance patterns” instead of fixed positions; better shift invariance
- Rotary Position embeddings (RoPE) allow relative positioning without any trainable parameters
  - Done inside self-attention, right after the linear projection and before computing attention scores
  - Unlike regular absolute embeddings which are baked/added into the input itself



Let:

- $\mathbf{x}_m \in \mathbb{R}^d$: token embedding at position m,
- $\mathbf{q}_m = W_q \mathbf{x}_m$: query vector,
- $\mathbf{k}_n = W_k \mathbf{x}_n$: key vector,
- d is even.

We split $\mathbf{q}_m$ and $\mathbf{k}_n$ into $d/2$ consecutive 2D subvectors and apply a position-dependent rotation to each subvector using a block-diagonal matrix $\mathbf{R}_{\Theta,m} \in \mathbb{R}^{d \times d}$:

$$\mathbf{q}_m^{\text{RoPE}} = \mathbf{R}_{\Theta,m} \mathbf{q}_m, \quad \mathbf{k}_n^{\text{RoPE}} = \mathbf{R}_{\Theta,n} \mathbf{k}_n$$

Where each 2D block in $\mathbf{R}_{\Theta,m}$ is a rotation matrix:

$$\begin{bmatrix} \cos(m\theta_i) & -\sin(m\theta_i) \\ \sin(m\theta_i) & \cos(m\theta_i) \end{bmatrix} \quad \text{for } i = 1, \ldots, d/2$$

and $\theta_i = 10000^{-2(i-1)/d}$

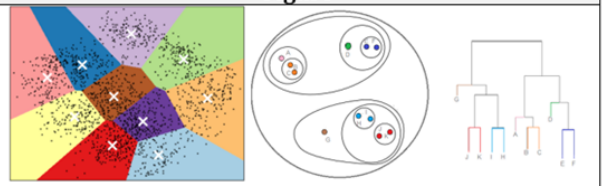
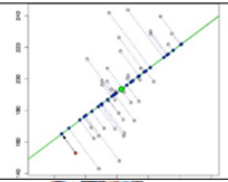
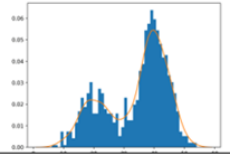
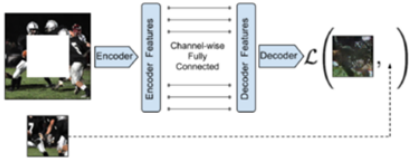
$$\text{score}_{m,n} = \langle \mathbf{q}_m^{\text{RoPE}}, \mathbf{k}_n^{\text{RoPE}} \rangle = \mathbf{q}_m^\top \mathbf{R}_{\Theta,m}^\top \mathbf{R}_{\Theta,n} \mathbf{k}_n = \mathbf{q}_m^\top \mathbf{R}_{\Theta,n-m} \mathbf{k}_n$$

Unsupervised & Self-Supervised Learning

At a high level, what is unsupervised learning? What are its advantages and disadvantages? How does it compare to Fully-supervised, Reinforcement, semi-supervised, and weakly supervised learning?

- In **unsupervised learning**, algorithms are not provided with any pre-assigned labels, and must self-discover any naturally occurring patterns in the training set.
  - Instead of learning the mapping x->y, we want to learn about x itself.
- In practice unsupervised learning can be naturally suited for:
  - **Freeform generative tasks** such as image/speech generation
  - **Statistical structure summarization/insight tasks** such as clustering, dimensionality reduction, or density estimation
  - **Pattern and representation learning**, e.g. through an auxiliary task which doesn't need human labels (i.e. **self-supervised learning**, a subset of unsupervised learning). Then, the representations can be used with supervision for a target downstream task such as classification, e.g. through fine-tuning, in a **semi-supervised fashion**.
- Comparison to other learning paradigms:
  - **Supervised learning**: usually naturally suited for recognition/classification/regression tasks where the desired output cannot be learned from unlabeled training data alone.
  - **Reinforcement learning:** Only numerical scores are available for each training example instead of detailed labels
  - **Semi-Supervised Learning:** Only a portion of the training data have been tagged
  - **Weakly Supervised learning:** Labels are given, but they are noisy and/or for an auxiliary task
- Advantages of no supervision:
  - Doesn't need a considerable amount of expert human labor for annotation
  - More data available to train on
  - More similar to how humans learn. Although we do get some label information, a child does all its learning unsupervised.
- Disadvantages of no supervision:
  - Too much training data can lead to slower convergence and increased computational requirements
  - Greater susceptibility to misleading artifacts, anomalies, and/or correlations in the data

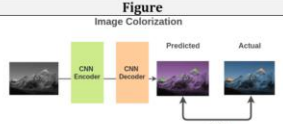
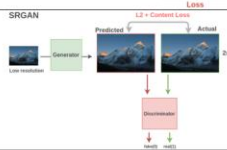
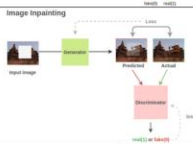
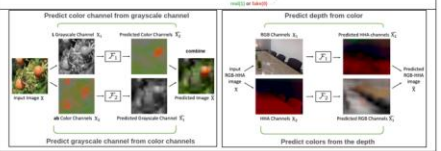
Give some examples and applications of unsupervised learning.

Overview of Common Unsupervised Learning Tasks/Approaches				
Unsupervised Task	**Description**	**Examples**	**Applications**	**Figure**
Clustering	Aims to partition many observations into clusters.	K-Means, Hierarchical Clustering	Useful for finding meaningful groups in data, or discovery of "prototype" elements to summarize the dataset	
Dimensionality Reduction	Tries to remove some of the dimensions in a dataset	PCA, SVD, Self organizing maps, Autoencoders	Data compression, denoising AEs	
Generation	Given a dataset, tries to learn how to sample "novel" realizations from it.	GANS, VAE	Picture/speech generation	
Density estimation	Want to learn the distribution of the data. Generally requires assumptions about the general family of distribution, and then parameters are estimated.	MLE, MAP, mixture distribution fitting	Exploratory data analysis (e.g. for skewness or multi modality), Outlier detection	
Representation learning	Aims to find mappings to a more information representation	Self-supervised learning auxiliary tasks like inpainting, jigsaw, or word completion; autoencoders	Finetuning with labeled data for downstream applications, in a semi-supervised fashion	

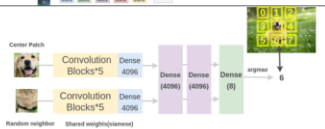
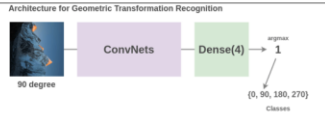
What is self-supervised learning? Explain some common methods based on reconstruction, “common sense”, and automatic labels.

- The goal of **self-supervised learning (SSL)** is to utilize auxiliary tasks in a way where we can generate virtually unlimited labels from our existing images, to learn useful representations. The resulting network can then be finetuned on a domain-specific downstream task.
- In a talk by Yann Lecun, he stated that self-supervised learning should be the bulk of the learning, in a cake analogy:
  - “If intelligence is a cake, the bulk of the cake is self-supervised learning, the icing on the cake is supervised learning, and the cherry on the cake is reinforcement learning (RL).”

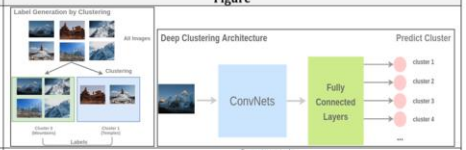
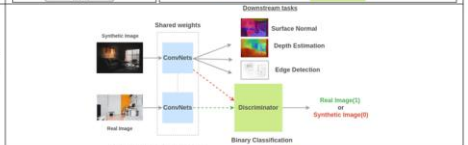
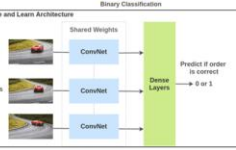
Reconstruction Based SSL

Auxiliary Task Name	Description	Figure
Image Colorization	What if we prepared pairs of (grayscale, colorized) images by applying grayscale to millions of images we have freely available?	
Image Superresolution	What if we prepared training pairs of (small, upscaled) images by downsampling millions of images we have freely available?	
Image Inpainting	What if we prepared training pairs of (corrupted, fixed) images by randomly removing part of images?	
Cross Channel Prediction	What if we predict one channel of the image from the other channel and combine them to reconstruct the original image? This could also be used for depth.	

Common-Sense Based SSL

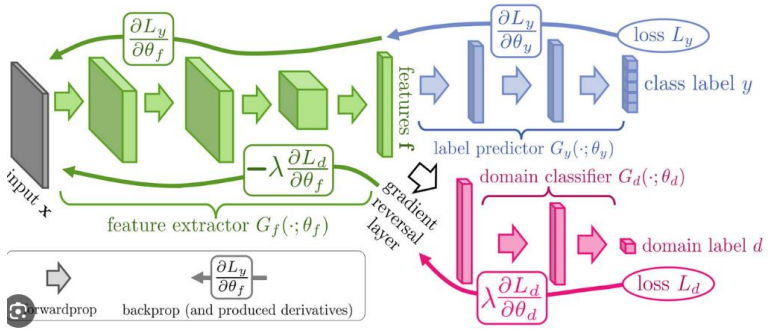
Auxiliary Task Name	Description	Figure
Image Jigsaw Puzzle	What if we prepared training pairs of (shuffled, ordered) puzzles by randomly shuffling patches of images?	
Context Prediction	What if we prepared training pairs of (image-patch, neighbor) by randomly taking an image patch and one of its neighbors around it from large, unlabeled image collection?	
Geometric Transformation Recognition	What if we prepared training pairs of (rotated-image, rotation-angle) by randomly rotating images by (0, 90, 180, 270) from large, unlabeled image collection?	

Automatic-Labels Based SSL

Auxiliary Task Name	Description	Figure
Image Clustering	What if we prepared training pairs of (image, cluster-number) by performing unsupervised clustering on large image collection?	
Synthetic Imagery	What if we prepared training pairs of (image, properties) by generating synthetic images using game engines and using domain adaptation to real images?	
Video Frame Order Verification	What if we prepared training pairs of (video frames, correct/incorrect order) by shuffling frames from videos of objects in motion?	

What are some approaches to Unsupervised Domain Adaptation?

1. **Domain-Adversarial Neural Networks (DANN)**: DANN introduces a domain discriminator network that is trained to distinguish between source and target data while the feature extractor network tries to confuse the domain discriminator. This encourages the feature extractor to learn domain-invariant representations.
2. **Adversarial Training**: Similar to DANN, adversarial training involves adding an adversarial loss term to the training objective. The network learns to minimize the discrepancy between source and target domains by adversarially training a domain discriminator.
3. **Feature Matching**: In this approach, the difference in feature distributions between the source and target domains is minimized. The model is trained to match the statistics of source and target domain features, encouraging domain-invariant representations.
4. **Maximum Mean Discrepancy (MMD)**: MMD measures the difference between the mean embeddings of the source and target domains in a high-dimensional space. By minimizing this discrepancy, the model learns domain-invariant features.
5. **Self-training**: Self-training involves using the source domain model to predict labels for unlabeled target domain data. These pseudo-labeled samples are then combined with the source domain data to train a new model. This process iterates to improve target domain performance.



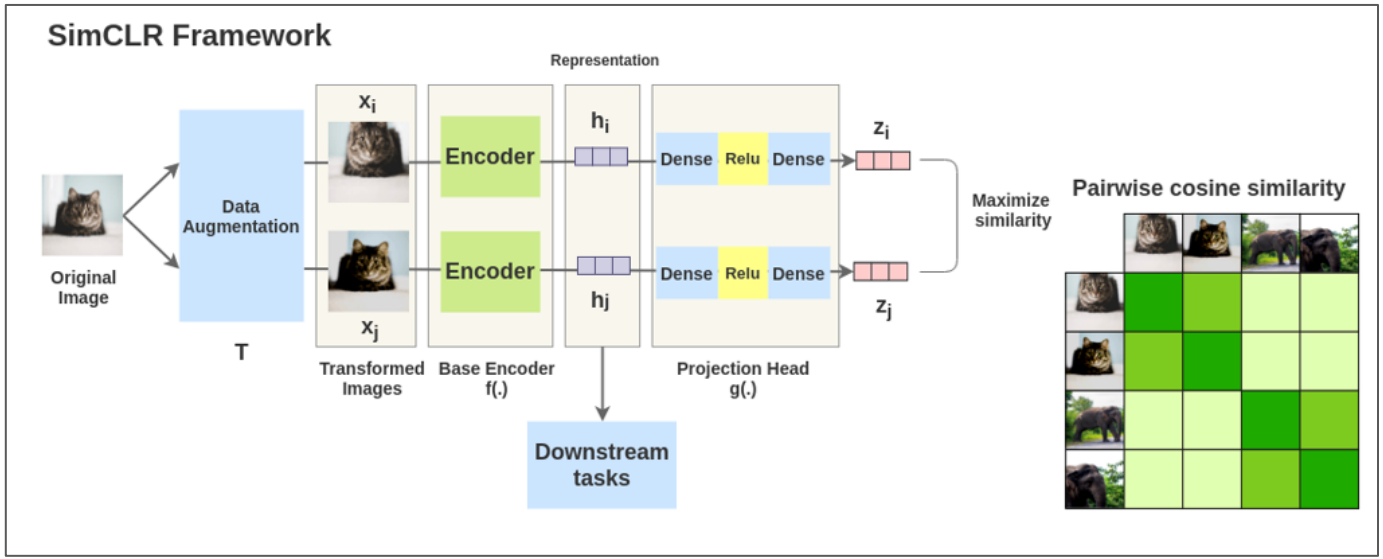
How can SSL be performed using contrastive learning?

- **SimCLR** (Google Brain, by Hinton, ICML 2020) was the first paper to show that SSL can match traditional supervised training
  - In the sense that it matches ResNet if you use the SSL-learned representations with a linear layer, trained all the labels.
  - Alternatively if you fine-tune (the whole network) with 1% labels, it gets 86% top-1 accuracy on ImageNet. In contrast, ResNet-50 with the same labels only achieves 48%.
- The main idea is to **use data augmentation transforms in an embedding contrastive learning** framework. The SSL part comes from the data augmentation to produce positive pairs, rather than something like class labels.
- N images are sampled in a batch, and only two augmentations are applied to each image to create pairs
  - Instead of sampling negative examples explicitly, given a positive pair, the other 2(N-1) augmented examples are treated as negative

The contrastive loss used is based on cross-entropy:

$$\ell_{i,j} = -\log \frac{\exp(\text{sim}(\boldsymbol{z}_i, \boldsymbol{z}_j)/\tau)}{\sum_{k=1}^{2N} \mathbb{1}_{[k \neq i]} \exp(\text{sim}(\boldsymbol{z}_i, \boldsymbol{z}_k)/\tau)}$$

- i, j are the positive pair in the minibatch
- τ is a temperature parameter
- sim is the cosine similarity function

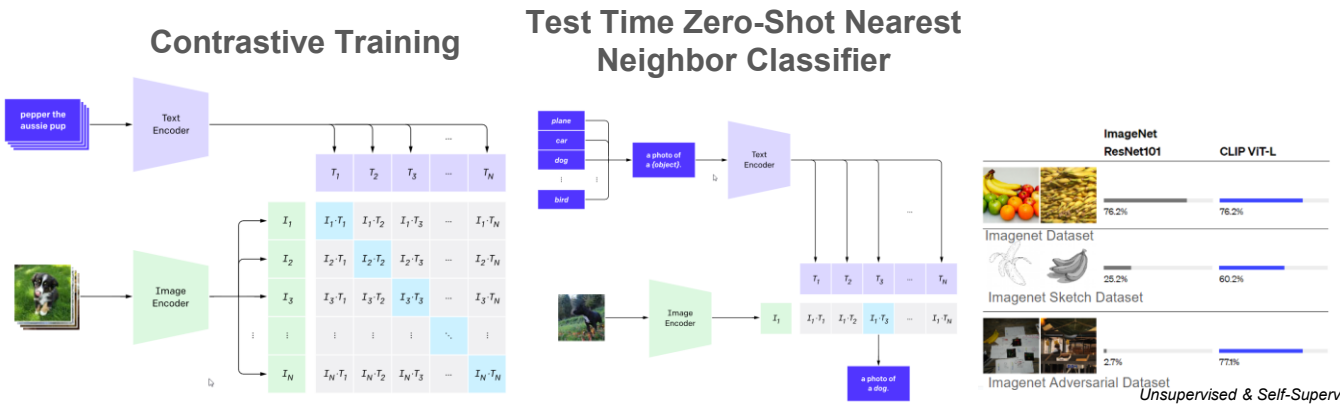


How does CLIP work?

- CLIP (Contrastive Language–Image Pre-training) learns a multimodal shared text/image embedding space from text-image pairs found on the internet
- Proxy Task: given image, predict which out of a set of 32,768 randomly sampled text snippets, was actually paired with it
- Given a minibatch of N text-image pairs, text & image encoders are trained to maximize cosine similarity of the N correct pairs, while minimizing for the N^2-N incorrect pairs, using a cross-entropy loss
- Authors found the proxy task of correctly selecting given captions was easier to learn & more well defined than just predicting exact words in caption
- Matches performance of ResNet on ImageNet “zero-shot”, without using any of the original labels
  - This is done by embedding text queries & using nearest neighbors
- Surpasses networks trained on imagenet on data distributions that are not seen on ImageNet

```
# image_encoder - ResNet or Vision Transformer
# text_encoder  - CBOW or Text Transformer
# I[n, h, w, c] - minibatch of aligned images
# T[n, l]       - minibatch of aligned texts
# W_i[d_i, d_e] - learned proj of image to embed
# W_t[d_t, d_e] - learned proj of text to embed
# t             - learned temperature parameter

# extract feature representations of each modality
I_f = image_encoder(I) #[n, d_i]
T_f = text_encoder(T)  #[n, d_t]

# joint multimodal embedding [n, d_e]
I_e = l2_normalize(np.dot(I_f, W_i), axis=1)
T_e = l2_normalize(np.dot(T_f, W_t), axis=1)

# scaled pairwise cosine similarities [n, n]
logits = np.dot(I_e, T_e.T) * np.exp(t)

# symmetric loss function
labels = np.arange(n)
loss_i = cross_entropy_loss(logits, labels, axis=0)
loss_t = cross_entropy_loss(logits, labels, axis=1)
loss   = (loss_i + loss_t)/2
```

Figure 3. Numpy-like pseudocode for the core of an implementation of CLIP.



How does DINO work?

- DINO (Distillation with No Labels) aims to learn rich, **general-purpose image features in an unsupervised way**, for various downstream applications
- Follows **student-teacher framework**, where both have the same vision transformer architecture but different weights
  - Given an image, teacher sees large global crops while student sees smaller zoomed-in views
  - Want to minimize cross-entropy between the softmaxed output embeddings of the teacher & student
  - Incentivizes learning of robust & invariant features
- Teacher's weights are a moving average of the student's weights; in a forward pass, its weights are not directly updated.
  - Teacher generally performs better
  - Intuition: teacher is like a slowly moving compass guide
- Tricks used to prevent collapse & degenerate solutions (notable cases are all uniform outputs, or only using one dimension):
  - **Sharpening**: The teacher's outputs are made peaky (via softmax with low temperature), which encourages diversity in the targets.
    - $P_t(x)^{(i)} = \frac{\exp\left(g_{\theta_t}(x)^{(i)}/\tau_t\right)}{\sum_{k=1}^{K} \exp\left(g_{\theta_t}(x)^{(k)}/\tau_t\right)}$
    - Here we choose a low value for $\tau_t > 0$ to encourage sharpening
    - Note that we don't have class supervision, so this helps to learn/infer proxy classes implicitly
  - **Centering**: The teacher's outputs are centered over the batch (mean-subtracted), encouraging more uniform-like distributions
    - If teacher logits are especially large in a direction of a dimension, softmax will create peaky results. By centering, we adjust for dominant values in those dimensions and bring it closer to zero, resulting in more uniform softmax outputs from the teacher
  - These two effects are opposites of one another, and empirically applying both balances them to prevent collapse
  - The EMA teacher with stop gradient itself is likely also an important regularization to prevent collapse. Would probably quickly collapse if everything was unfrozen.
- Positive feedback loop, where both get better over time
  - Starts with blurry noise correlations, which gradually become more semantic/informative over time
- Ultimately student is used for downstream applications

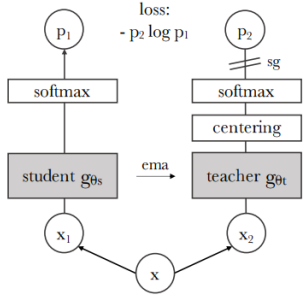


Figure 2: **Self-distillation with no labels.** We illustrate DINO in the case of one single pair of views (x_1, x_2) for simplicity. The model passes two different random transformations of an input image to the student and teacher networks. Both networks have the same architecture but different parameters. The output of the teacher network is centered with a mean computed over the batch. Each networks outputs a K dimensional feature that is normalized with a temperature softmax over the feature dimension. Their similarity is then measured with a cross-entropy loss. We apply a stop-gradient (sg) operator on the teacher to propagate gradients only through the student. The teacher parameters are updated with an exponential moving average (ema) of the student parameters.

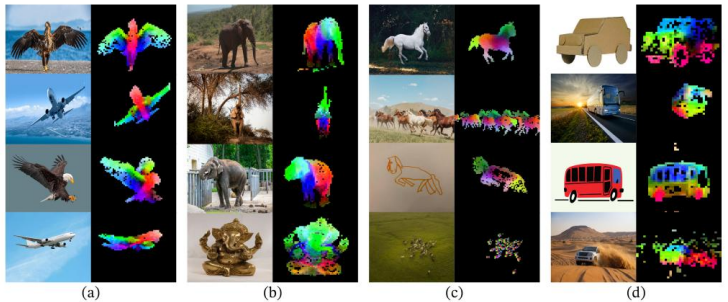


Figure 1: **Visualization of the first PCA components.** We compute a PCA between the patches of the images from the same column (a, b, c and d) and show their first 3 components. Each component is matched to a different color channel. Same parts are matched between related images despite changes of pose, style or even objects. Background is removed by thresholding the first PCA component.

Semi-Supervised Learning

At a high level, what is semi-supervised learning, and what are the common approaches/assumptions?

- Goal of semi-supervised learning is to learn with some data that's labeled, and some that is not.
  - Note that regardless of the training method, supervised pretraining on a different dataset/task is still generally helpful.

- Some concrete approaches:
  - **Psuedolabeling / self-learning**
  - **Self-supervised (eg generative, contrastive) pre-training**
  - **Consistency regularization**
    - Encourages the model to output similar predictions for an input and its perturbations
  - **Entropy minimization**
    - Trains the model to make confident (low-entropy) predictions on unlabeled data.

- Several hypotheses have been discussed in literature to support certain design decisions in semi-supervised learning methods:
  - **Smoothness Assumptions:** If two data samples are close in a high-density region of the feature space, their labels should be the same or very similar.
  - **Cluster Assumptions**: The feature space has both dense regions and sparse regions. Densely grouped data points naturally form a cluster. Samples in the same cluster are expected to have the same label. This is a small extension of H1.
  - **Low-density Separation Assumptions**: The decision boundary between classes tends to be located in the sparse, low density regions, because otherwise the decision boundary would cut a high-density cluster into two classes, corresponding to two clusters, which invalidates H1 and H2.
  - **Manifold Assumptions**: The high-dimensional data tends to locate on a low-dimensional manifold. This enables us to learn a more efficient representation for us to discover and measure similarity between unlabeled data points.