Low Level / Classical Vision

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

- 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:
    - $R = R_c^T$
    - $T = -R_c^T C$

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

$$\left[\begin{array}{c|c} R & t \\ \hline \mathbf{0} & 1 \end{array}\right] = \left[\begin{array}{c|c} R_c & C \\ \hline \mathbf{0} & 1 \end{array}\right]^{-1}$$
$$= \left[\left[\begin{array}{c|c} I & C \\ \hline \mathbf{0} & 1 \end{array}\right]\left[\begin{array}{c|c} R_c & 0 \\ \hline \mathbf{0} & 1 \end{array}\right]\right]^{-1}$$
$$= \left[\begin{array}{c|c} R_c & 0 \\ \hline \mathbf{0} & 1 \end{array}\right]^{-1}\left[\begin{array}{c|c} I & C \\ \hline \mathbf{0} & 1 \end{array}\right]^{-1}$$
$$= \left[\begin{array}{c|c} R_c^T & 0 \\ \hline \mathbf{0} & 1 \end{array}\right]\left[\begin{array}{c|c} I & -C \\ \hline \mathbf{0} & 1 \end{array}\right]$$
$$= \left[\begin{array}{c|c} R_c^T & -R_c^T C \\ \hline \mathbf{0} & 1 \end{array}\right]$$

- The **intrinsic camera matrix** $\boldsymbol{K}$ 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 / how "zoomed in" the view is. Usually $f_x = f_y$ for square pixels
  - **Skew** $\boldsymbol{s}$. Shifts objects at an angle, typically zero
  - **principal point offset**, c_x, c_y, affects xy location of object. Eg for a 640 x 480 camera, this would generally be 320, 240

$$K = \begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

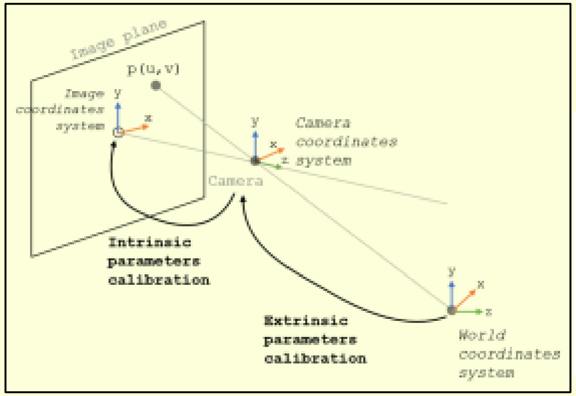
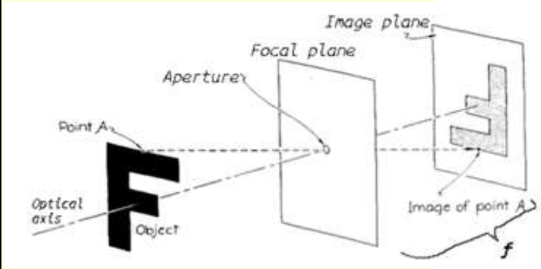
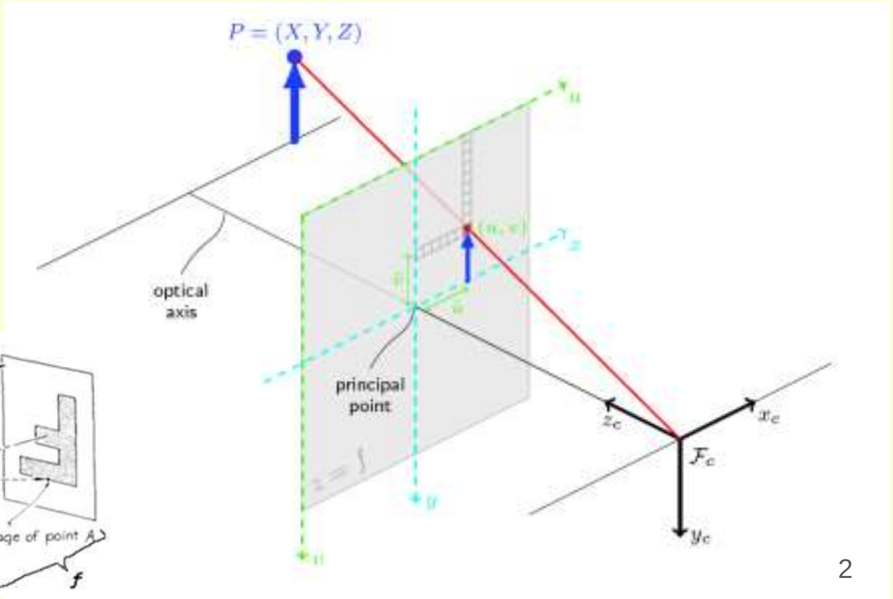


Image plane
p(u,v)
Image coordinates system
y
x
Camera coordinates system
z
Camera
Intrinsic parameters calibration
Extrinsic parameters calibration
World coordinates system





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)

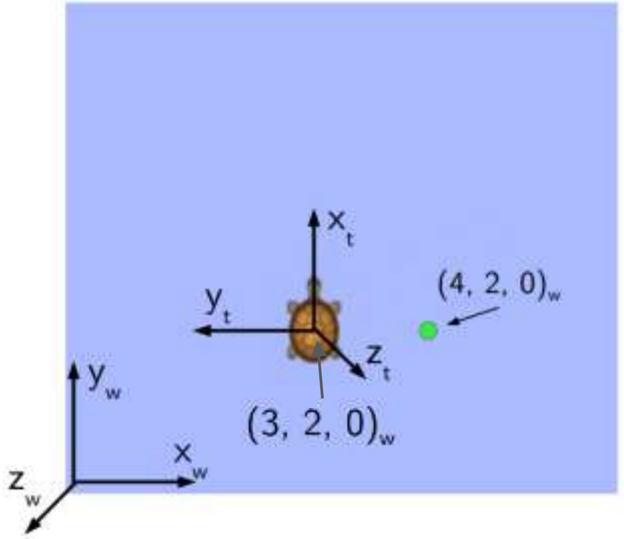
What is (4,2,0)w in turtle coordinates?

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

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

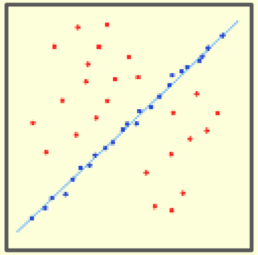
$$R_{w2t} = \begin{bmatrix} 0 & 1 & 0 & -2 \\ -1 & 0 & 0 & 3 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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



What 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.

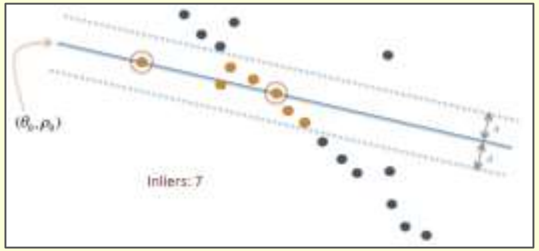


RANSAC psuedocode:

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

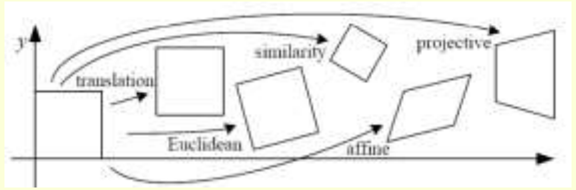
Until convergence:

- *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)

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.

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

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

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

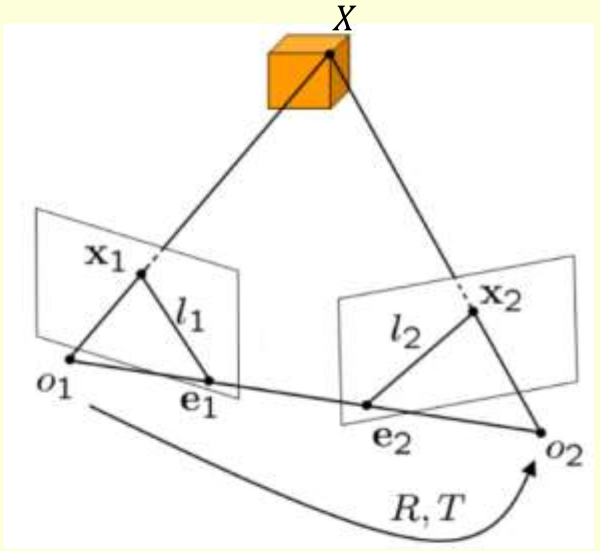
    Where $\theta_{ij}=1$ if point j is visible in image i, else 0 and π denotes the perspective projection function
  - Typically solved using Levenberg-Marquardt or gradient descent
- **Loop closure (Optional)**
  - Detect when the camera revisits a previously seen location, and do a pose graph optimization

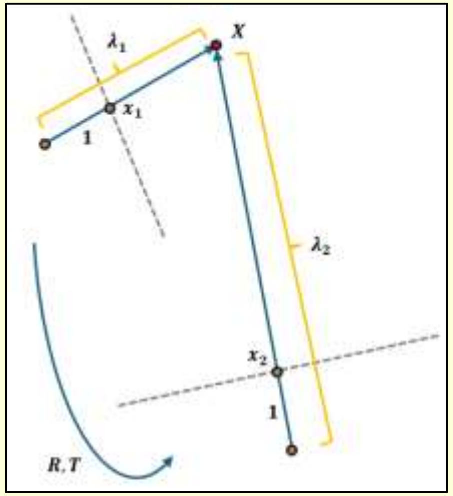
Derive the epipolar constraint & describe its geometric interpretations.

- The right figure illustrates the geometry of a 3D point & 2 cameras:
  - $\boldsymbol{o_1, o_2}$**: Optical centers** of each camera
  - R, T: relative rotation & translation between the cameras
  - $\boldsymbol{x_1, x_2}$**: Projections** of 3D point X onto two images.
    - In normalized image coordinates, ie in camera frame after applying intrinsics with K^{-1}
  - $\boldsymbol{e_1, e_2}$**: Epipoles**, which are the intersection of the line (o_1, o_2) with each image plane
  - $(\boldsymbol{o_1, o_2, X})$**: Epipolar plane**; there is one such plane for each 3D point X
  - (o_1, o_2): **Baseline vector** between the two cameras
  - $\boldsymbol{l_1, l_2}$**: 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":

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

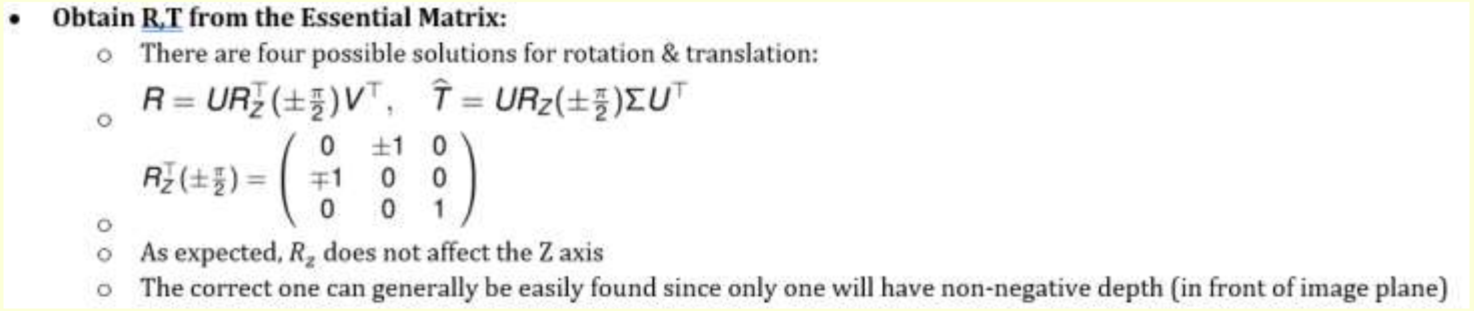
  - For any vector v, $u \times v = \hat{u}v$; so this is another way to express cross products
- Then, the Epipolar constraint can be derived as follows:
  - $\lambda_1 x_1 = X, \quad \lambda_2 x_2 = RX + T$
  - $\lambda_2 x_2 = R(\lambda_1 x_1) + T$
  - $\lambda_2 \hat{T} x_2 = \lambda_1 \hat{T} R x_1$, 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 $\mathrm{R}x_1$ 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 (fullrank)
- 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:
  - $R = UR_Z^\top(\pm\frac{\pi}{2})V^\top, \quad \hat{T} = UR_Z(\pm\frac{\pi}{2})\Sigma U^\top$
  - $R_Z^\top(\pm\frac{\pi}{2}) = \begin{pmatrix} 0 & \pm 1 & 0 \\ \mp 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$
  - 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)

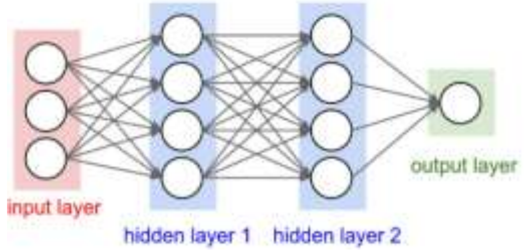
How does the fundamental differ from the essential matrix? How can you recover the rotation & translation from the fundamental matrix?

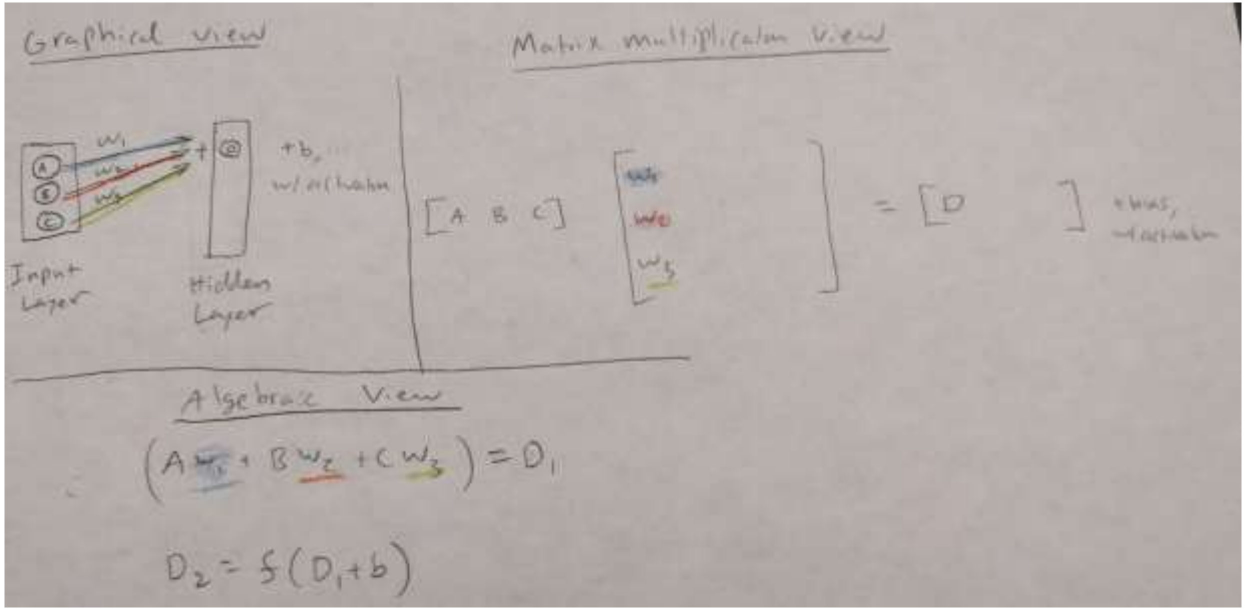
- The **fundamental matrix** F relates Epipolar constraints without knowing calibration: $\boldsymbol{x_2^T F x_1 = 0}$
  - Here, x_1, x_2 are in pixel coordinates; intrinsics not applied
  - Has the same geometric intuitions as $x_2^T E x_1 = 0$; it’s just that F here also encodes intrinsics information
  - Can be estimated the same way using the 8 point algorithm
- If you assume constraints on intrinsics (zero skew, square pixel, known principal point, $K_1 = K_2$), you can estimate K_1, K_2 from F using the **Kruppa equations**
- Once you know the intrinsics, the essential matrix can be recovered by $\boldsymbol{E = K_2^T F K}$ and then you can recover the rotation & translation using the same process

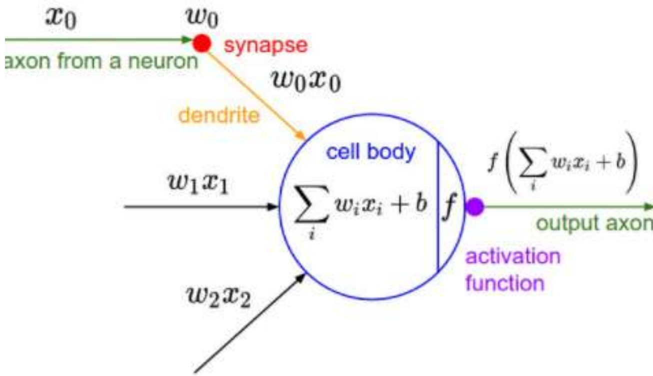
Deep Learning Fundamentals

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.

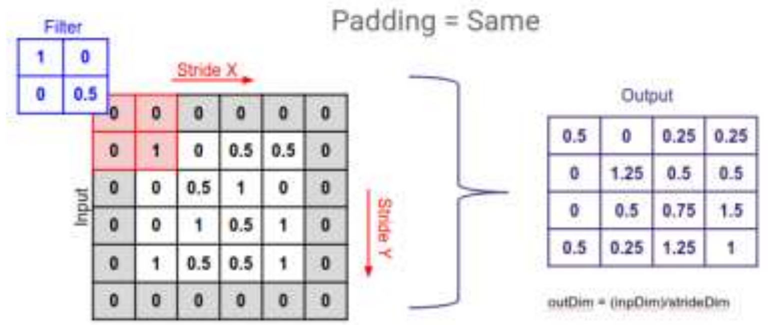






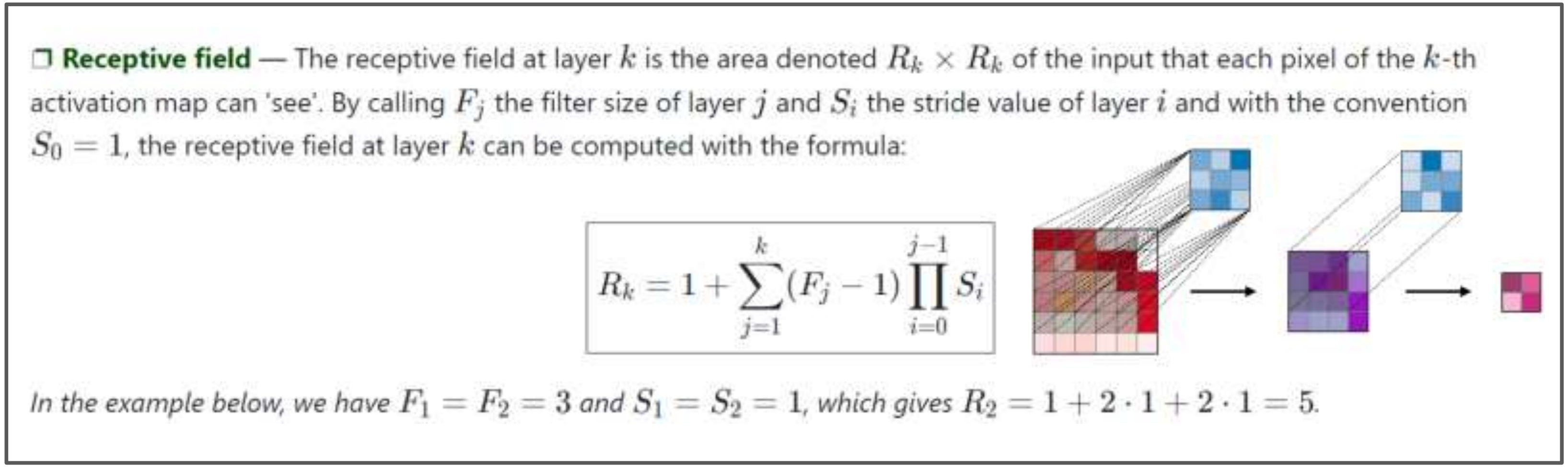
What is the output depth, padding, stride, and filter size of a conv layer?

- **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**



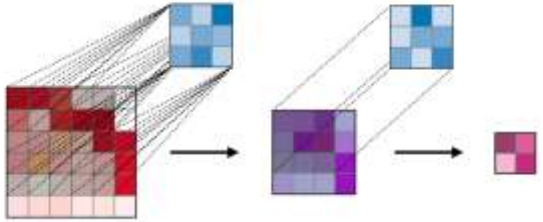
What makes convnets translationally invariant? What is the “receptive field”?

- 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:

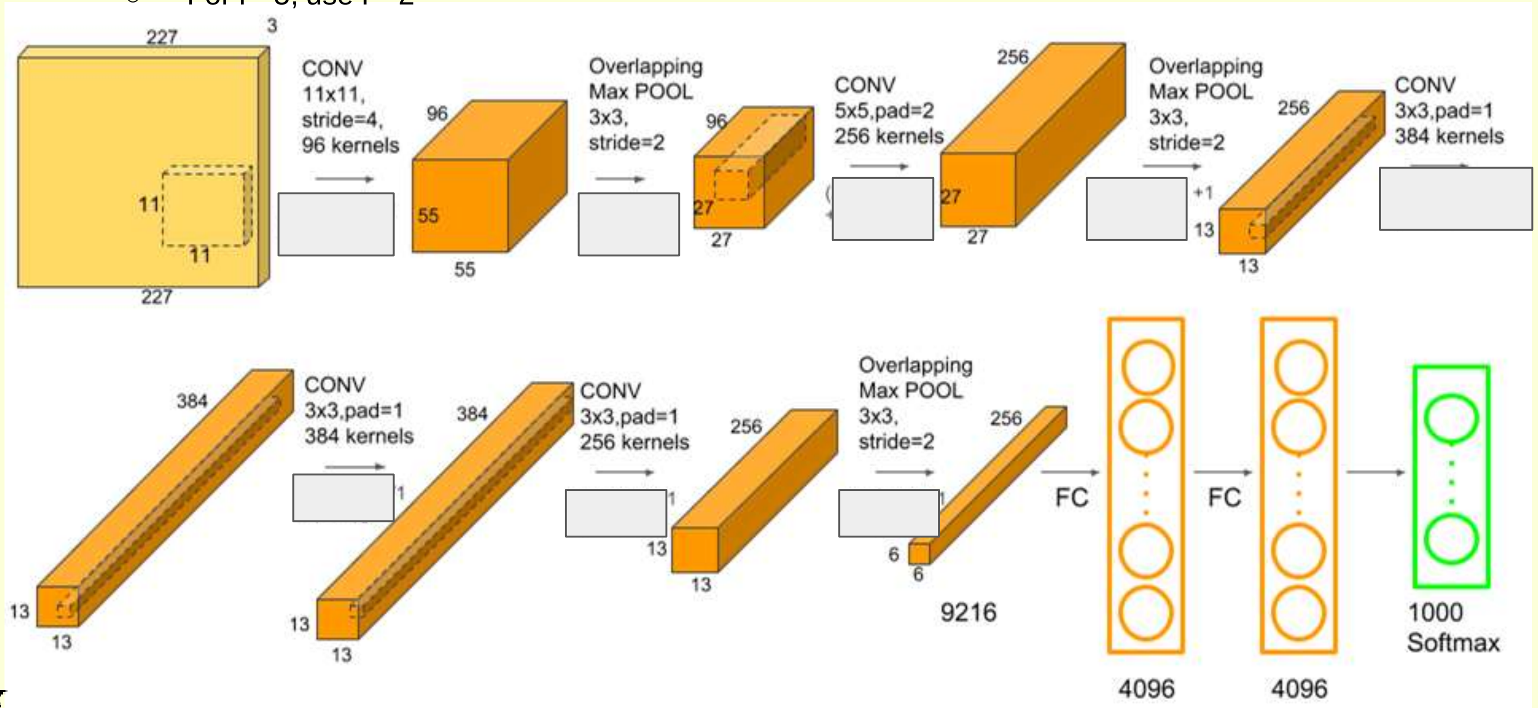
$$\boxed{R_k = 1 + \sum_{j=1}^{k}(F_j - 1)\prod_{i=0}^{j-1} S_i}$$



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?

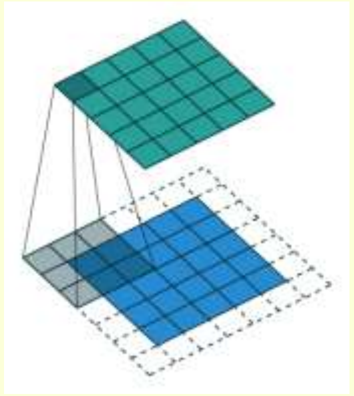
- Conv layer: Output spatial size will be D = ((W+2P-F)/S) + 1; output feature map size will be O x D x D
- Pooling layer: ((W-F)/S)+1; output depth stays the same
- To maintain spatial sizes:
  - For F=3, use P=1
  - For F=5, use P=2



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

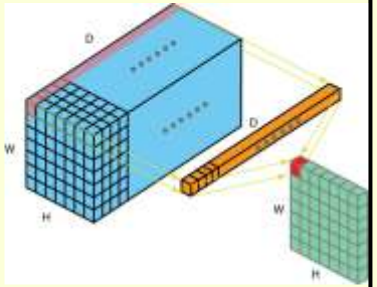
Regular Convolutions

- 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



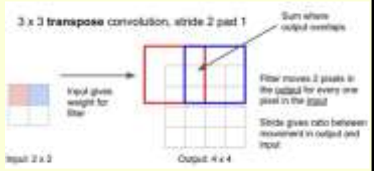
1x1 Convolutions

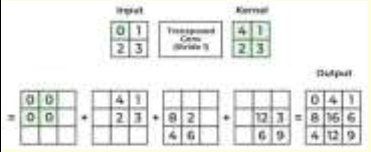
- Same as regular conv, but 1 x 1 x c filters. This keeps the spatial dimensions, but changes the channels
- Useful for dim. reduction, and introducing nonlinearities
- Output array from each filter after convolving with input can be thought of as a weighted sum of the input feature map's channels



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.

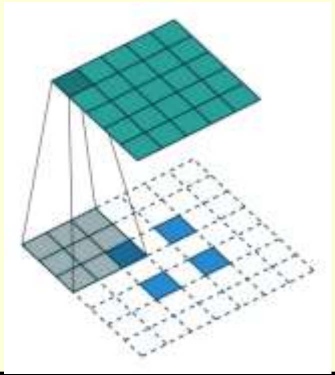

3 x 3 **transpose** convolution, stride 2 pad 1
Sum where output overlaps
Input gives weight for filter
Filter moves 2 pixels in the output for every one pixel in the input
Stride gives ratio between movement in output and input
Input: 2 x 2
Output: 4 x 4



Input	
0	1
2	3

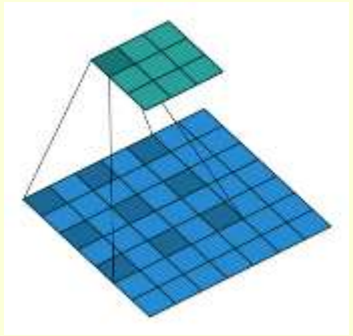
Kernel	
4	1
2	3

$$= \begin{bmatrix} 0 & 0 & \\ 0 & 0 & \\ & & \end{bmatrix} + \begin{bmatrix} & 4 & 1 \\ & 2 & 3 \\ & & \end{bmatrix} + \begin{bmatrix} & & \\ 8 & 2 & \\ 4 & 6 & \end{bmatrix} + \begin{bmatrix} & & \\ & 12 & 3 \\ & 6 & 9 \end{bmatrix} = \begin{bmatrix} 0 & 4 & 1 \\ 8 & 16 & 6 \\ 4 & 12 & 9 \end{bmatrix}$$



Atrous/Dilated Conv.

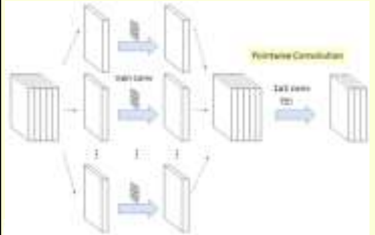
- Adds gaps in the filter when convolving, increasing the field of view even with the same number of parameters.
- However, this does not spatially upsample

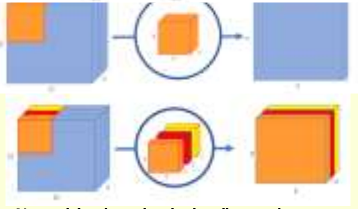


Depthwise Separable Conv. (From MobileNet)

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.



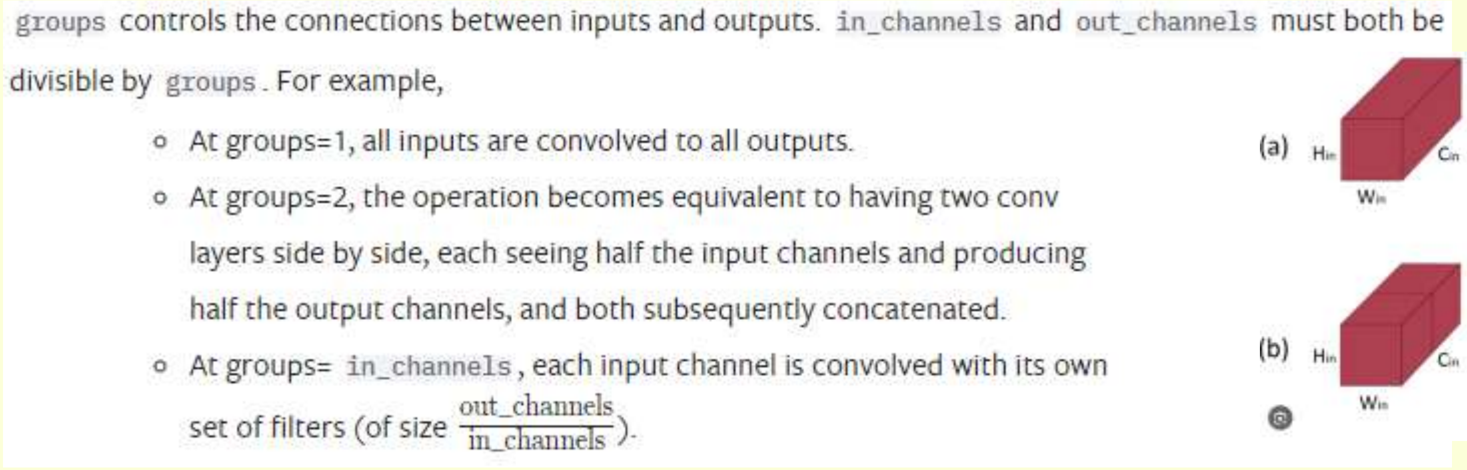


Normal (top) vs depthwise (bottom) conv

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.

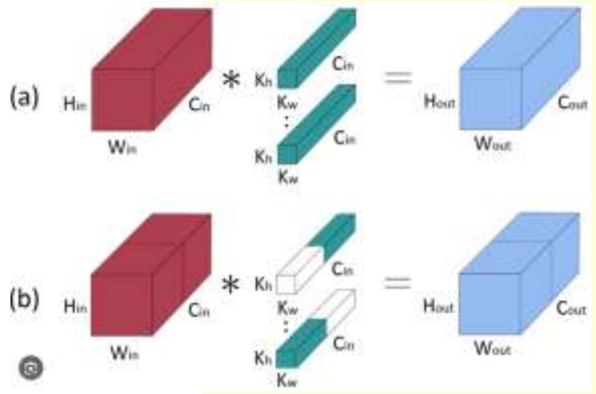
$$\frac{depthwise}{regular} = \frac{KKHWM + HWMN}{KKHWMN} = \frac{HWM(KK + N)}{KKHWMN} = \frac{KK + N}{KKN} = \frac{1}{N} + \frac{1}{K^2}$$

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 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

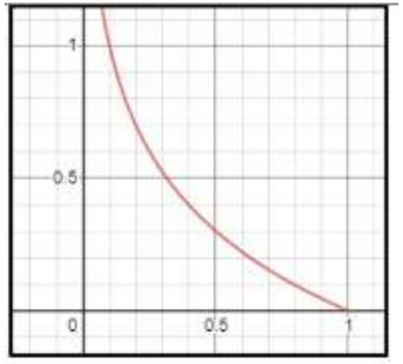
The softmax is often paired with the **cross-entropy loss** (aka **negative log likelihood**), which operates on vectors of class probabilities.

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

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

$f = -\log(x)$



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

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

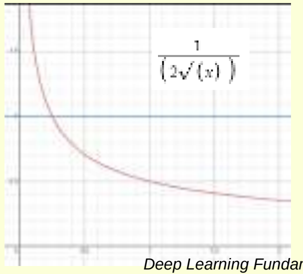
- **Adam** (Learning rate per parameter)
  - Keeps an (weighted) exponential moving average of the gradient and the squared gradient (ie, the first two moments)
  - The weighted gradient is used, and the learning rate is divided by the square root of the squared gradient.
  - Lower variance is more stable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

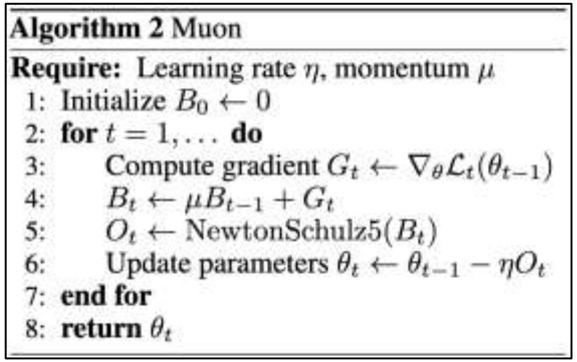
- torch.optim.lr_scheduler.**StepLR**
  - Decays learning rate by multiplication with *gamma* every *step_size* epochs.
- torch.optim.lr_scheduler.**MultiStepLR**
  - Decays learning rate by multiplication with *gamma* every time a specified milestone epoch is reached.
- torch.optim.lr_scheduler.**ReduceLROnPlateau**
  - 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)

How does the Muon optimizer work?

- **Muon ("MomentUm Orthogonalized by Newton–Schulz")** is a geometry driven NN optimizer, specialized for 2D matrix parameters (eg in hidden/linear layers)
- Takes SGD-momentum updates and semi-orthogonalizes them using an algorithm using Netwon-Schultz
  - An orthogonal matrix's columns & rows form an orthogonal set; they're pairwise perpendicular & have unit length
  - Thus, intuitively this helps train more efficiently by giving importance to "rare directions" that are often overlooked by Adam or SGD outputs
- Has set training speed records, and scales to larger LLMs



```
Algorithm 2 Muon
Require: Learning rate $\eta$, momentum $\mu$
1: Initialize $B_0 \leftarrow 0$
2: for $t = 1, \ldots$ do
3:     Compute gradient $G_t \leftarrow \nabla_\theta \mathcal{L}_t(\theta_{t-1})$
4:     $B_t \leftarrow \mu B_{t-1} + G_t$
5:     $O_t \leftarrow \text{NewtonSchulz5}(B_t)$
6:     Update parameters $\theta_t \leftarrow \theta_{t-1} - \eta O_t$
7: end for
8: return $\theta_t$
```

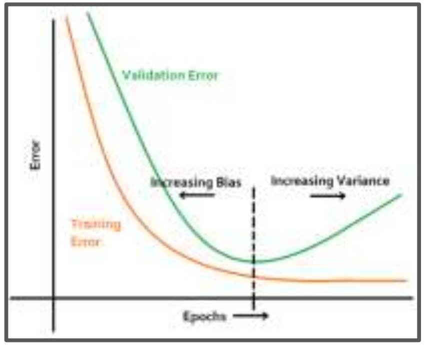
Describe some basic regularization techniques (5).

- **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)

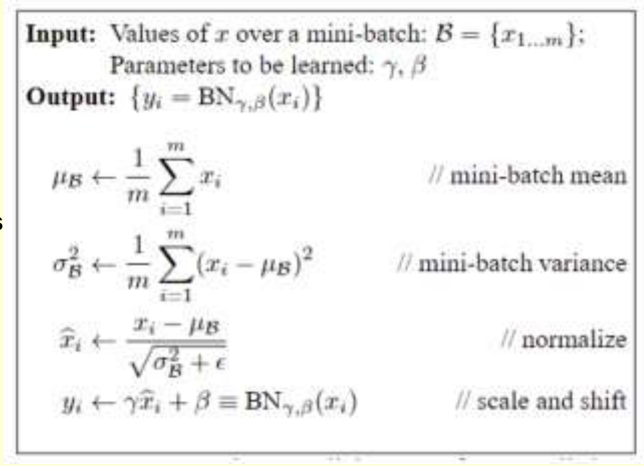
$$\widetilde{E}(\mathbf{w}) = E(\mathbf{w}) + \frac{\lambda}{2}\mathbf{w}^2$$

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

- **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: $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

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

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

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

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

Implementation Notes:

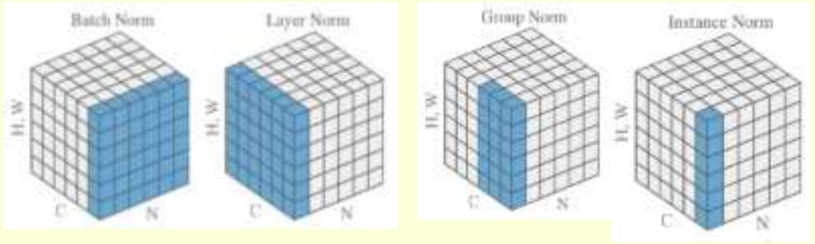
- Makes training **faster and more stable**
- **For convolutional layers**: we want different elements of the same conv filter to be normalized in the same way. Thus, the normalization is performed at the per-channel level, over the batch.
- Batch normalization adds two trainable parameters γ and β to 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.

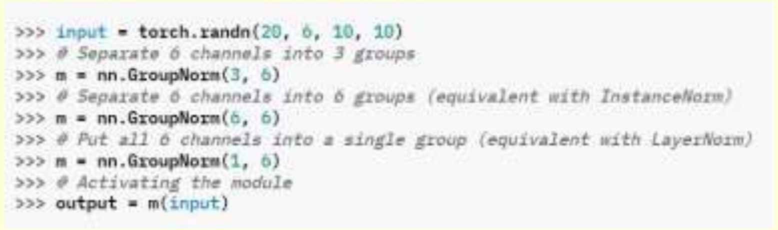
Several theories on why batchnorm works:

- **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 become unnecessary

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.
- Instead, these norm approaches calculates the statistics per-minibatch sample, instead of per-channel across all minibatches
- **Layer Norm** is very common for LLMs
  - Normalization computed on an individual per-token basis; mean & variance over token hidden dimension
  - Different intuition/dynamics:
  - **BatchNorm**: “Make this feature have the same distribution across the batch.”
  - **LayerNorm**: “Make this token’s features have a stable scale before we do math with them.”
  - LayerNorm still computes the mean/variance from the *current* input vector at inference. BatchNorm freezes and uses pre-computed stats from training.
- **Group norm** may be able to exploit/utilize some internal/inherent structures within features
- **Instance norm** seems niche and useful to speed up certain generative tasks



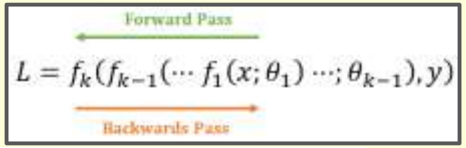


```
>>> input = torch.randn(20, 6, 10, 10)
>>> # Separate 6 channels into 3 groups
>>> m = nn.GroupNorm(3, 6)
>>> # Separate 6 channels into 6 groups (equivalent with InstanceNorm)
>>> m = nn.GroupNorm(6, 6)
>>> # Put all 6 channels into a single group (equivalent with LayerNorm)
>>> m = nn.GroupNorm(1,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

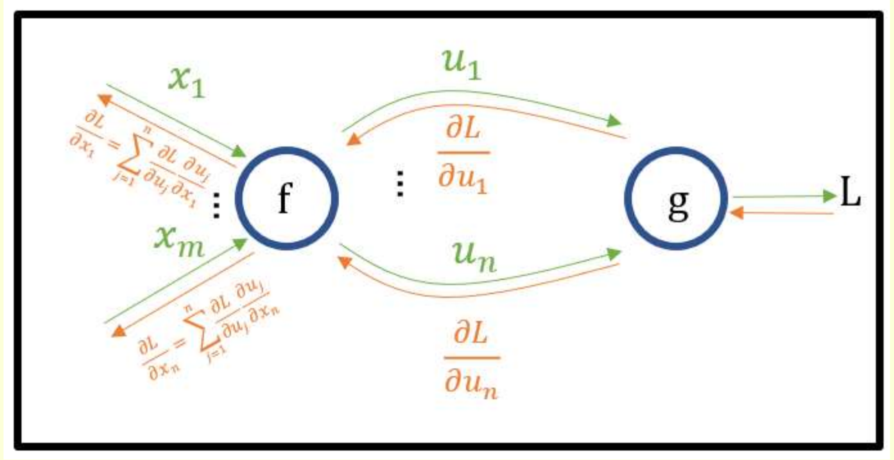


$$L = f_k(f_{k-1}(\cdots f_1(x;\theta_1)\cdots;\theta_{k-1}),y)$$

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 $f_i\colon \mathbb{R}^m \rightarrow \mathbb{R}^n$.
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 $g\colon \mathbb{R}^n \rightarrow \mathbb{R}$.
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.


x_1
$\frac{\partial L}{\partial x_1} = \sum_{j=1}^{n} \frac{\partial L}{\partial u_j}\frac{\partial u_j}{\partial x_1}$
x_m
$\frac{\partial L}{\partial x_n} = \sum_{j=1}^{n} \frac{\partial L}{\partial u_j}\frac{\partial u_j}{\partial x_n}$
f
u_1
$\frac{\partial L}{\partial u_1}$
u_n
$\frac{\partial L}{\partial u_n}$
g
L

"Upstream Gradient"
(based on later layers)

"Local Gradient"
(based on Jacobian)

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

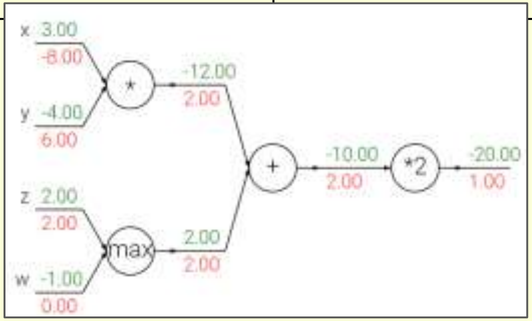
Impact of x_i on L wrt all outputs u_j are added to get overall impact

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)

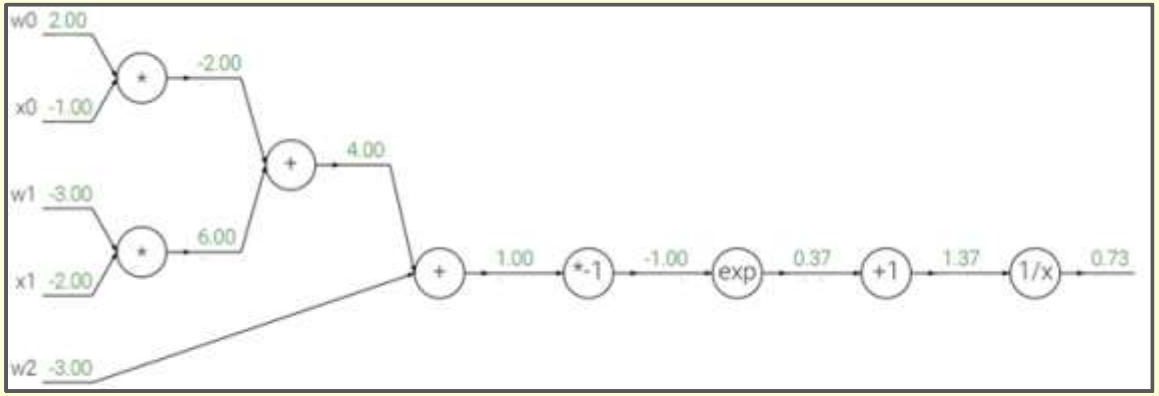
Describe how the following functions/gates propagate the upstream gradient during backpropagation: add, max, multiply

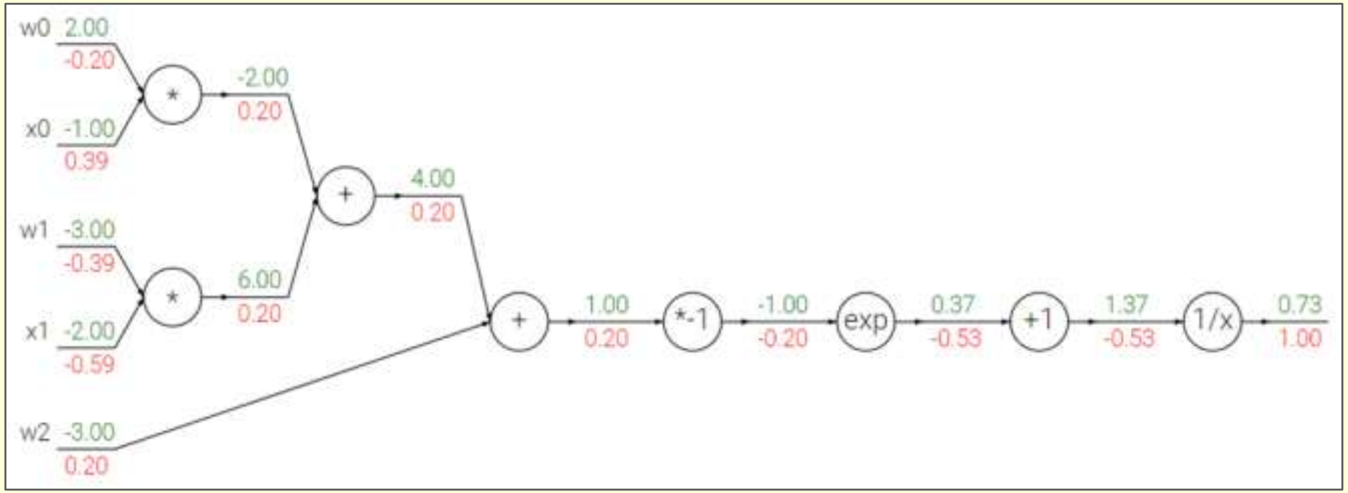
Gate	Behavior	Intuition
Add	Adds all the upstream gradient on its output and distributes that sum equally to all of its inputs. This follows from the fact that the local gradient for the add operation (e.g. x+7) is simply +1.0	Forwards gradients, unchanged.
Max	Distributes the gradient (unchanged) to 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).



<u>Complete the following computational graph for backpropagation.</u>

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

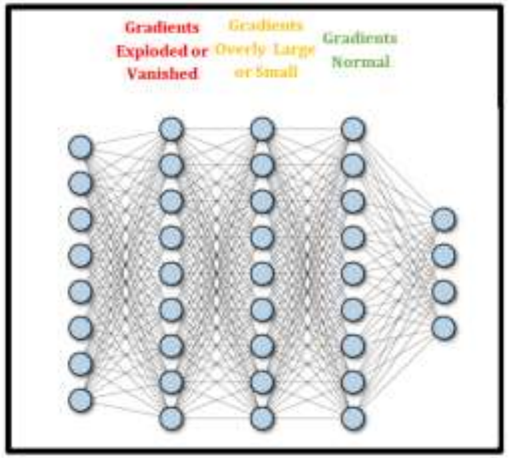




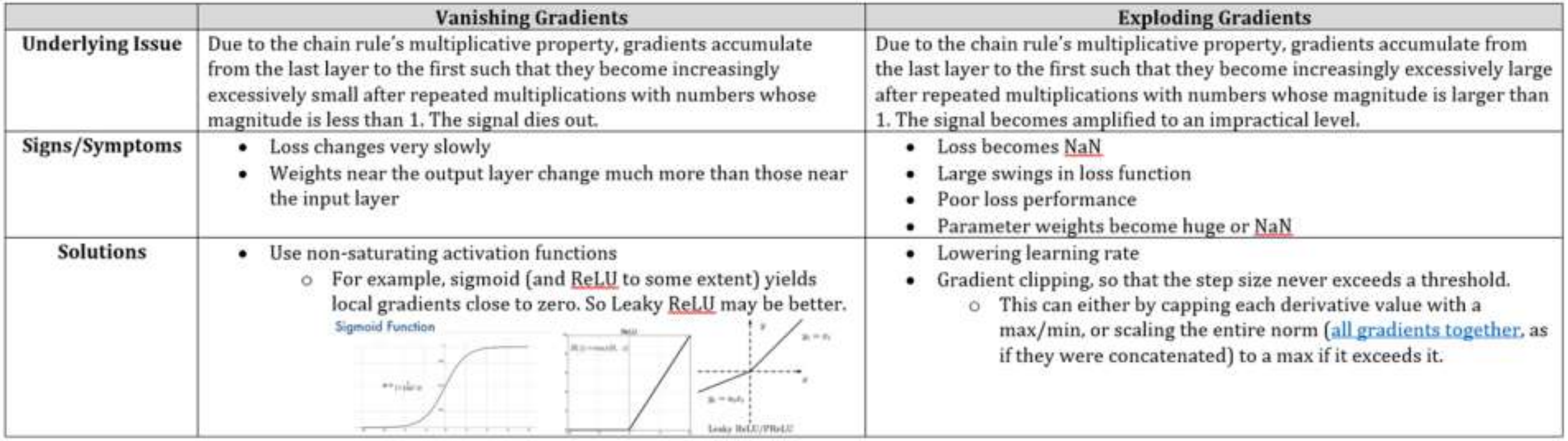
What is the “straight-through estimator” and when is it used?

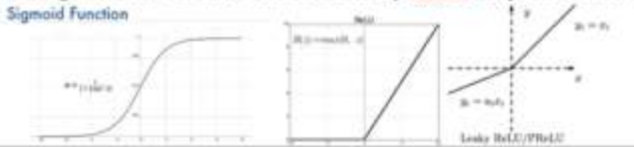
- Method used to approximate gradient during backwards pass when dealing with operations in the forwards-pass that are non-differentiable or whose gradients are difficult to compute
- For the purposes of the backwards pass, replaces the function by the identity function when computing the gradients with 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?


Gradients Exploded or Vanished
Gradients Overly Large or Small
Gradients Normal

- 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

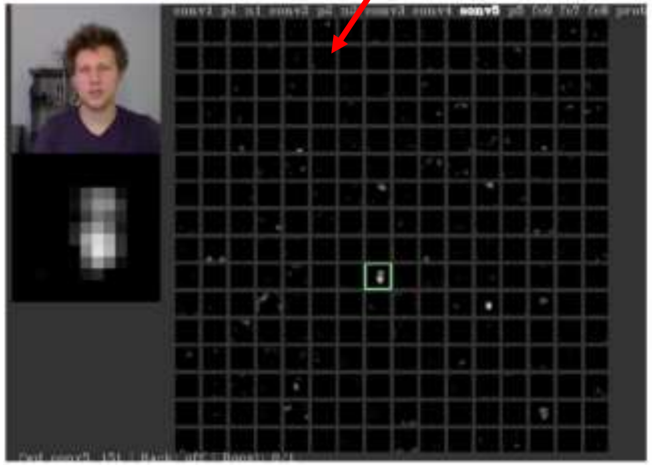
What are some sources of randomness/stochasticity in neural networks? Why is this a good thing?

- 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


conv1 p1 n1 conv2 p2 n2 conv3 conv4 conv5 p5 fc6 fc7 fc8 prob

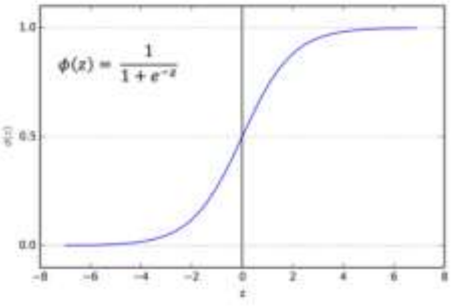


Fig: Sigmoid Function

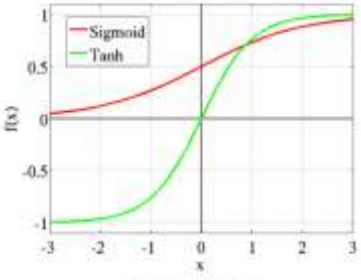
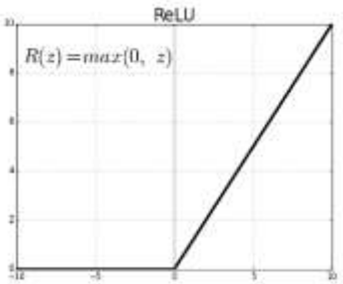
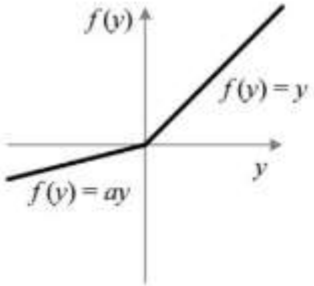


Fig: tanh vs Logistic Sigmoid

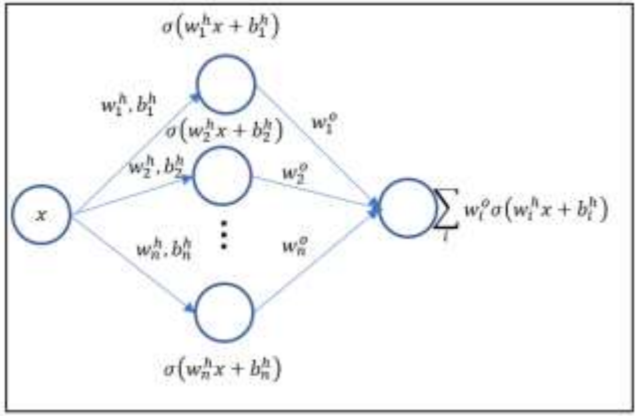




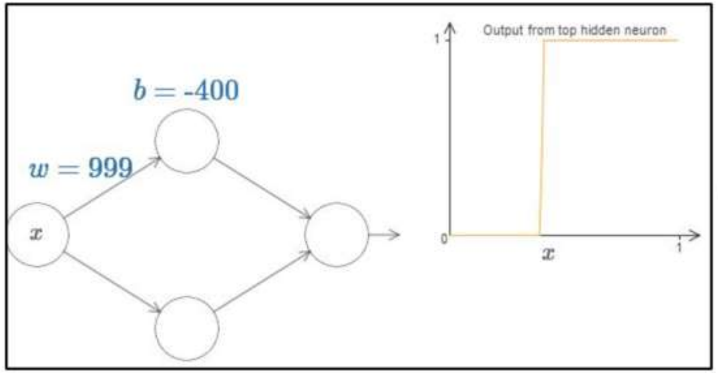
Leaky ReLU (usually a is around 0.01)

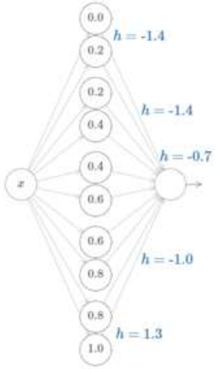
What is the universal approximation theorem, and what are its limitations for practical applications?

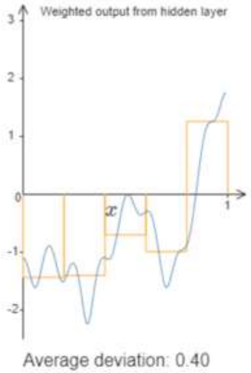
- A result that states that a FC network with only one hidden layer can approximate any function to arbitrary precision, given enough width (hidden neurons)
- At a high level, this is because sigmoid can represent step functions, and when enough are combined, can approximate any function
- In practice, 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

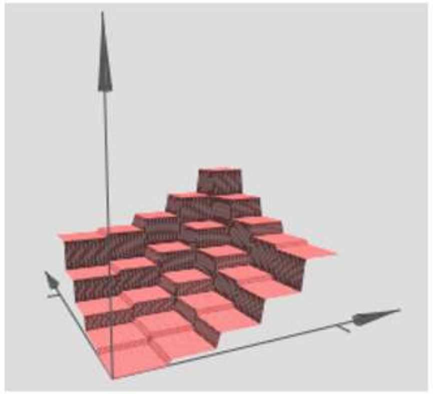


$\sigma(w_1^h x + b_1^h)$
w_1^h, b_1^h
$\sigma(w_2^h x + b_2^h)$
w_1^o
w_2^h, b_2^h
w_2^o
x
$\sum_i w_i^o \sigma(w_i^h x + b_i^h)$
w_n^h, b_n^h
w_n^o
$\sigma(w_n^h x + b_n^h)$



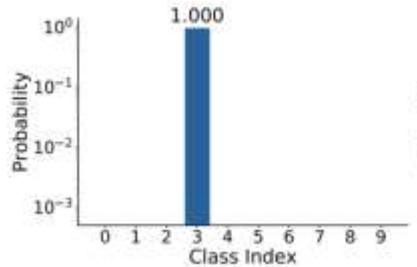




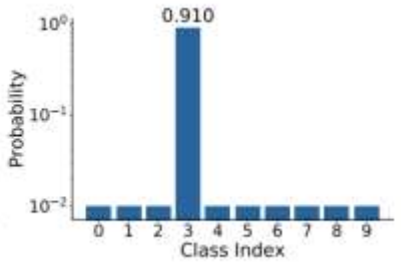


What is label smoothing?

- **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

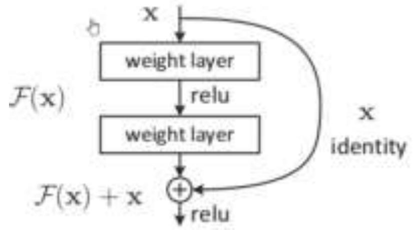


(b) LS

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


x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

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

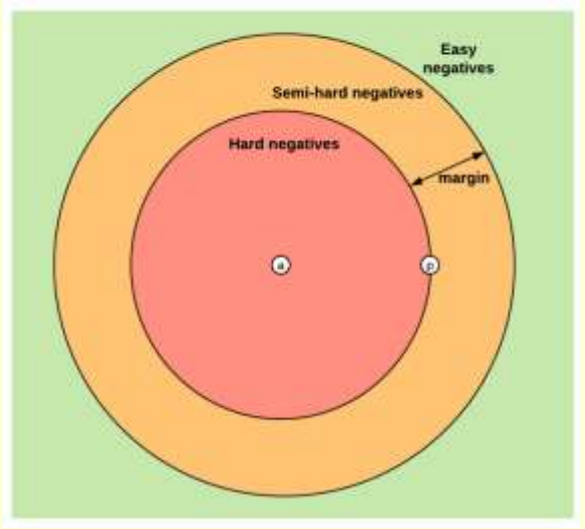
- **Embeddings for classification** are useful in cases where we have a variable number of classes (not fixed), for example face verification.
- **Embeddings for retrieval** is a natural fit, where you embed your test query and use k-nn in the embedding space to retrieve the top k entries.
- The **triplet 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

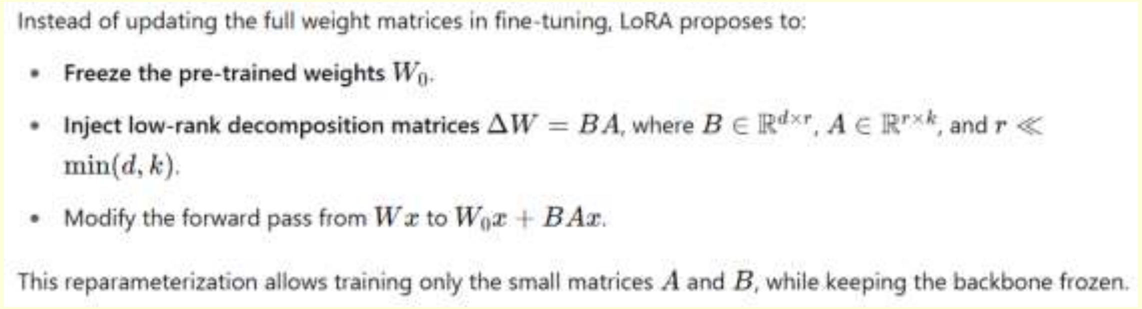
- There are N^3 possible triplets; for efficiency, we want to select good triplets to learn from.
- In FaceNet, they use random **semi-hard negatives**, recomputing after each epoch. There are 3 categories, given a fixed anchor and positive and relative to the negative:

- **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:

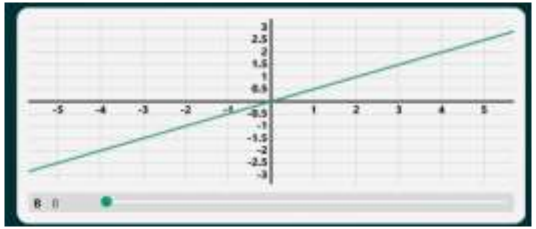
- **Freeze the pre-trained weights** W_0.
- **Inject low-rank decomposition matrices** $\Delta W = BA$, where $B \in \mathbb{R}^{d\times r}$, $A \in \mathbb{R}^{r\times k}$, and $r \ll \min(d, k)$.
- Modify the forward pass from Wx to $W_0x + BAx$.

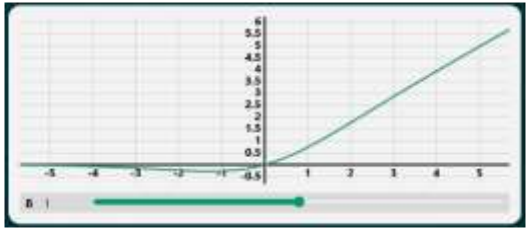
This reparameterization allows training only the small matrices A and B, while keeping the backbone frozen.

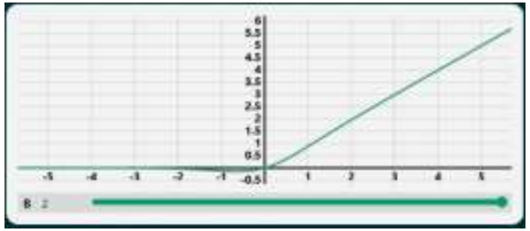
- This is applied to self-attention layers: query & value projection matrices W_q, W_v
- Recall that the rank of a matrix roughly tells you how much unique information it contains & its degrees of freedom. Note that LLMs are already heavily over-parameterized and contain general knowledge
  - Adapting 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?

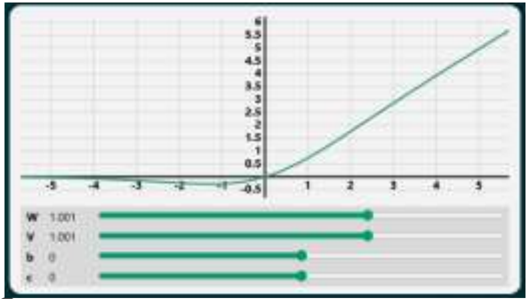
- **Swish Activation**: $swish(x) = x * sigmoid(\beta x)$
  - β is a learnable parameter; when $\beta = 0$, becomes linear; when $\beta \to \infty$, becomes ReLU

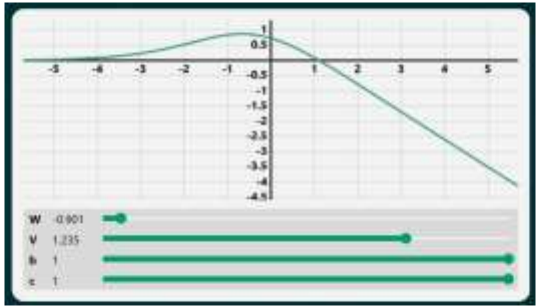

β 0

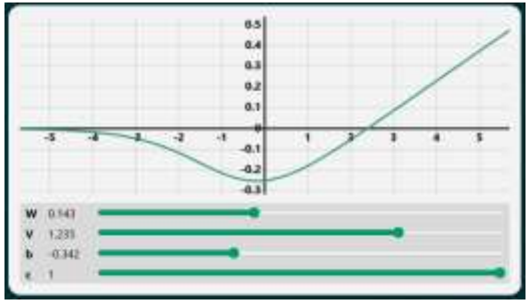

β 1


β 2

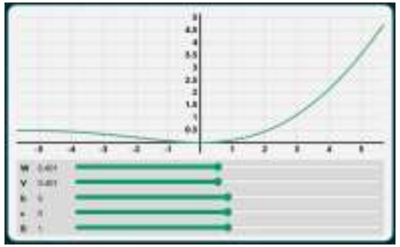
- **GLU (Gated Linear Unit) Activation**: $GLU(x) = (Wx + b) * sigmoid(Vx + c)$
  - W, V, b, c all learnable parameters
  - Can be interpreted as a linear transformation over x, times the probability of activating that neuron

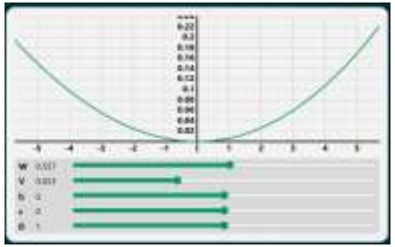

W 1.001
V 1.001
b 0
c 0

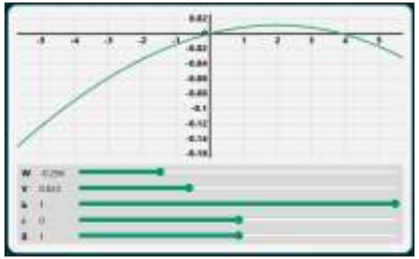

W -0.801
V 1.235
b 1
c 1

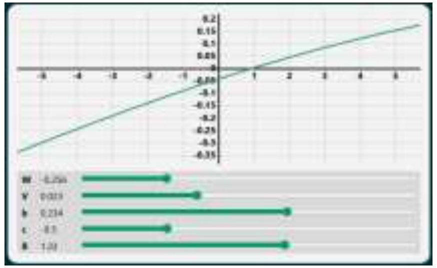

W 0.143
V 1.235
b -0.342
c 1

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


W [illegible]
V [illegible]
b 0
c 0
β 1


W [illegible]
V [illegible]
b 0
c 0
β 1


W [illegible]
V [illegible]
b 1
c 0
β 1


W [illegible]
V [illegible]
b 0.234
c [illegible]
β [illegible]

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

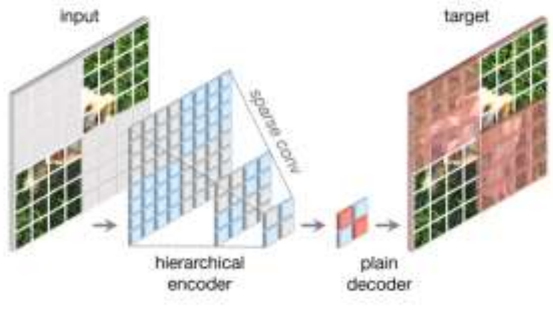


Figure 2. **Our FCMAE framework.** We introduce a fully convolutional masked autoencoder (FCMAE). It consists of a sparse convolution-based ConvNeXt encoder and a lightweight ConvNeXt block decoder. Overall, the architecture of our autoencoder is asymmetric. The encoder processes only the visible pixels,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

Unsupervised & Self-Supervised Learning

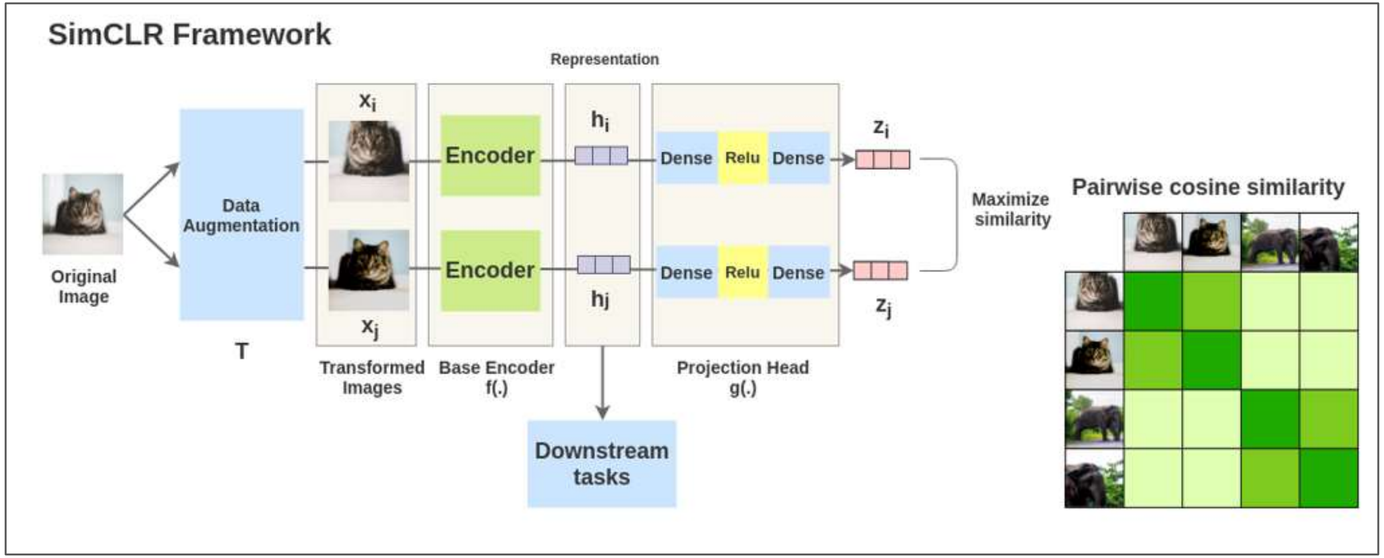
How can SSL be performed using contrastive learning?

- **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:

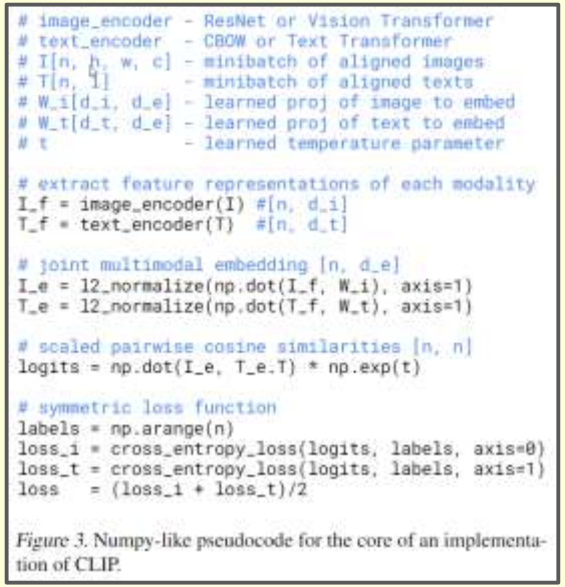
$$\ell_{i,j} = -\log \frac{\exp(\mathrm{sim}(\boldsymbol{z}_i, \boldsymbol{z}_j)/\tau)}{\sum_{k=1}^{2N} \mathbb{1}_{[k \neq i]} \exp(\mathrm{sim}(\boldsymbol{z}_i, \boldsymbol{z}_k)/\tau)}$$

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



How does CLIP work?

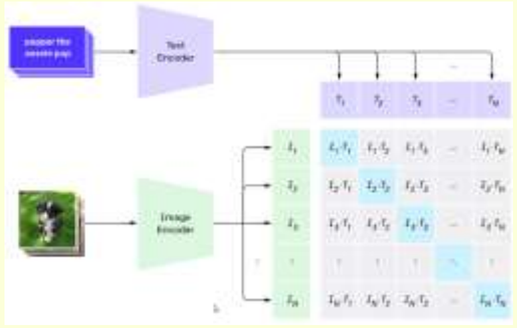
- CLIP (Contrastive Language–Image Pre-training) learns a multimodal shared text/image embedding space from text-image pairs found on the internet
- Proxy Task: given image, predict which out of a set of 32,768 randomly sampled text snippets, was actually paired with it
- Given a minibatch of N text-image 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



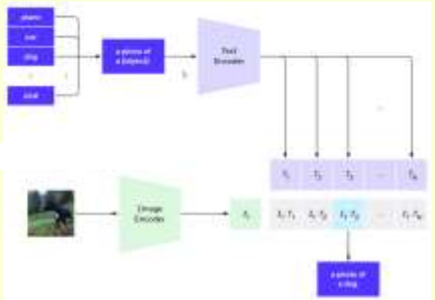
```
# image_encoder - ResNet or Vision Transformer
# text_encoder  - CBOW or Text Transformer
# I[n, h, w, c] - minibatch of aligned images
# T[n, l]       - minibatch of aligned texts
# W_i[d_i, d_e] - learned proj of image to embed
# W_t[d_t, d_e] - learned proj of text to embed
# t             - learned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

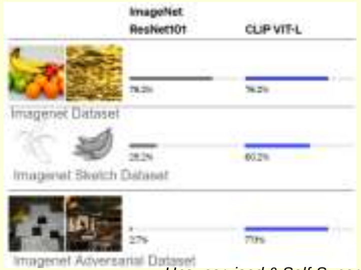
Figure 3. Numpy-like pseudocode for the core of an implementation of CLIP.

Contrastive Training



Test Time Zero-Shot Nearest Neighbor Classifier





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
- DinoV2 is an incremental improvement that has better data, small augmentation/architecture tweaks

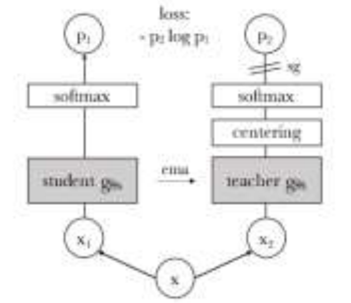


Figure 2: **Self-distillation with no labels.** We illustrate DINO in the case of one single pair of views (x_1, x_2) for simplicity. The model passes two different random transformations of an input image to the student and teacher networks. Both networks have the same architecture but different parameters. 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.

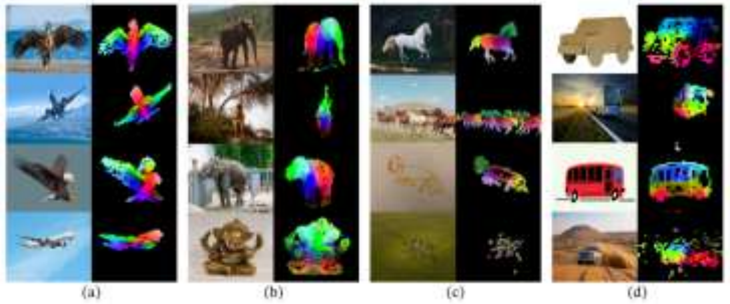


Figure 1: **Visualization of the first PCA components.** We compute a PCA between the patches of the images from the same column (a, b, c and d) and show their first 3 components. Each component is matched to a different color channel. Same parts are matched between related images despite changes of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

Semi-Supervised Learning

Question Goes Here

- Answer goes here