Low Level / Classical Techniques in Vision And Image Processing

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
  - T is a 3x1 vector representing the world origin in camera coordinates
  - R and T is a bit unintuitive to specify, so often we use:
    - R_c, a 3x3 matrix where columns represent the camera axes in world coordinates
    - C, a 3x1 vector representing the location of the camera center in world coordinates
    - R_c^T = R, which is the inverse of R_c
    - T = -R_c^TC
  - Another option is to use the “look_at” function to get R and t
- The **intrinsic camera matrix** represents the internal parameters of the camera: focal length, skew, and principal point. It can be found using camera calibration
- With these matrices, you specify:
  - Mapping from 3D point in world to the 2D pixel coordinate in image
  - Mapping from 2D pixel coordinate to a ray in 3D (with depth, you can further specify the exact point on that ray)

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

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

$$K = \begin{bmatrix} \alpha_x & \gamma & u_0 & 0 \\ 0 & \alpha_y & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

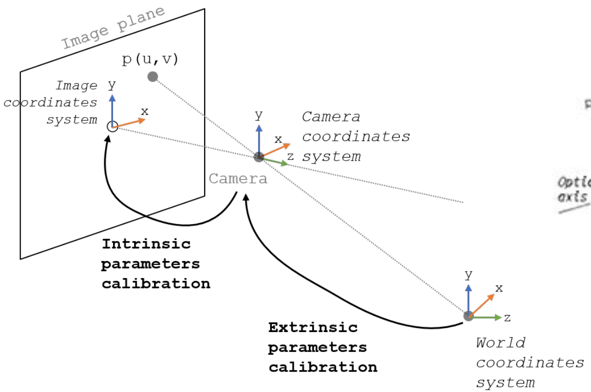
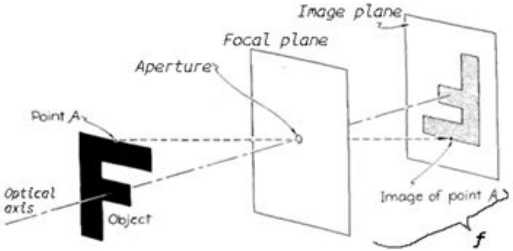


Image plane
p(u,v)
Image coordinates system
y
x
Camera coordinates system
y
x
z
Camera
Intrinsic parameters calibration
Extrinsic parameters calibration
y
x
z
World coordinates system



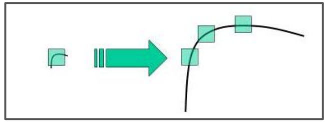
3D point -> 2D point (w=1)

$$\begin{bmatrix} x \\ y \\ w \end{bmatrix} = \underbrace{\begin{bmatrix} \mathbf{K} \end{bmatrix}}_{\text{intrinsics}} \underbrace{\left[\begin{array}{c|c} \mathbf{R} & \mathbf{t} \end{array}\right]}_{\text{inverse pose}} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

2D point -> 3D Ray (w=1, assume undistort is identity)

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = w \cdot \underbrace{\begin{bmatrix} \mathbf{R}^\top \end{bmatrix} \begin{bmatrix} \mathbf{K}^{-1} \end{bmatrix} \begin{bmatrix} \text{undistort}_x(x') \\ \text{undistort}_y(y') \\ 1 \end{bmatrix}}_{\text{ray direction}} \underbrace{- \begin{bmatrix} \mathbf{R}^{-1} \end{bmatrix} \begin{bmatrix} \mathbf{t} \end{bmatrix}}_{\text{camera center}}$$

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**

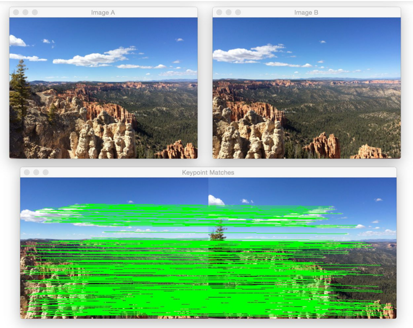
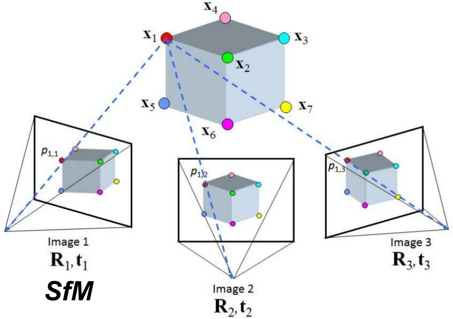
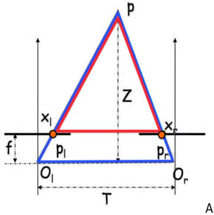


Image Stitching



SfM



Similar triangles:

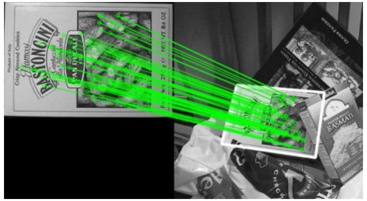
$$\frac{T}{Z} = \frac{T + x_l - x_r}{Z - f}$$

$$\boxed{Z = \frac{f \cdot T}{x_r - x_l}}$$

$$x = \frac{f \cdot X}{Z} + p_x$$

And if I know Z, I can compute X and Y, which gives me the point in 3D

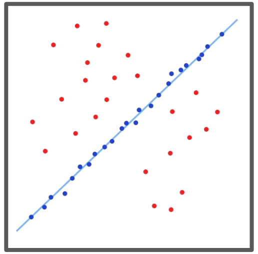
Depth Estimation



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.

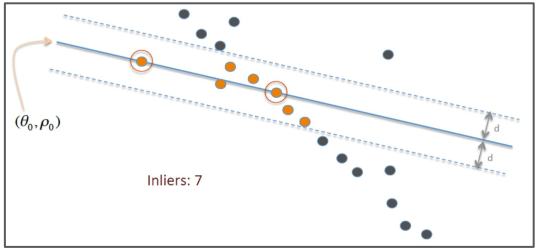


RANSAC psuedocode:

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

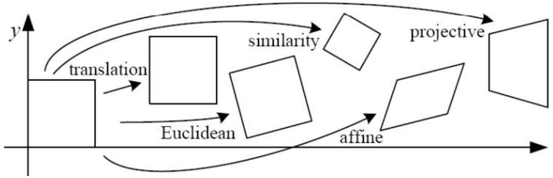
Until convergence:

- *Select a 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 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.

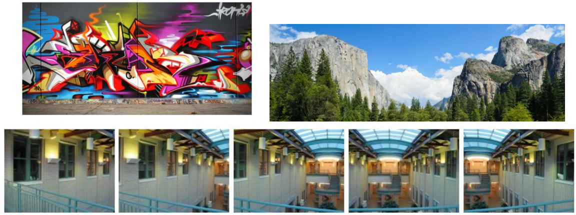
- A homography is a isomorphism between two vector spaces; ie, representable by a nonsingular matrix.
- Homographies are viable either of the following are true:
  - The scene is planar or approximately planar (ie, the scene is very far or has small relative depth variation)
  - The scene is captured 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.

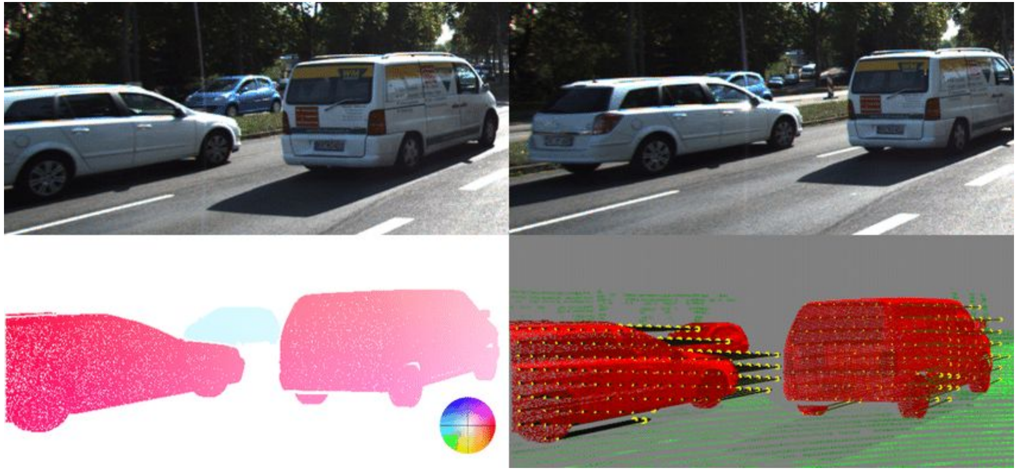
Left is Planar; Right is Approx Planar due to distance; Bottom is captured under cam. Rotation only



At a high level, what is optical flow?

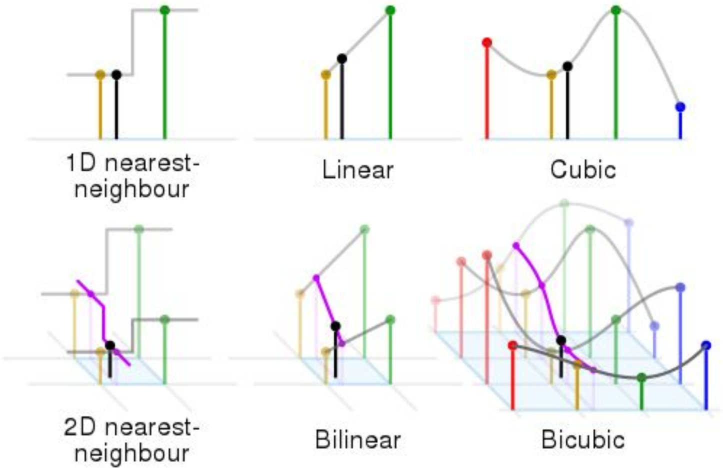
In both cases, the goal is to understand the motion of a scene.

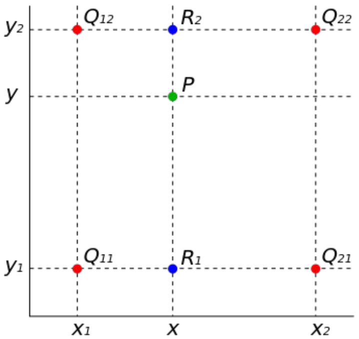
- **Feature based optical flow** is sparse, and establishes correspondences between two frames using characteristic features
  - Examples include SIFT
- **Dense optical flow** establishes correspondences between two frames applied to all pixels
  - A classical way to solve this is the Lucas-kanade algorithm, which is based on energy minimization

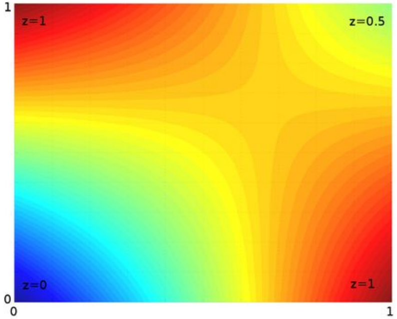


What is nearest-neighbors interpolation vs bilinear interpolation?

- In bilinear interpolation you interpolate in one direction (e.g. x), then in the second direction (e.g. y), by taking weighted averages
- In nearest neighbor interpolation, instead of weighted averages, you just pick the label of the nearest point.



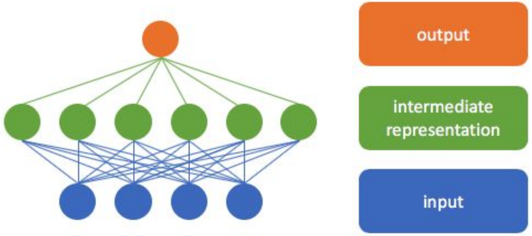


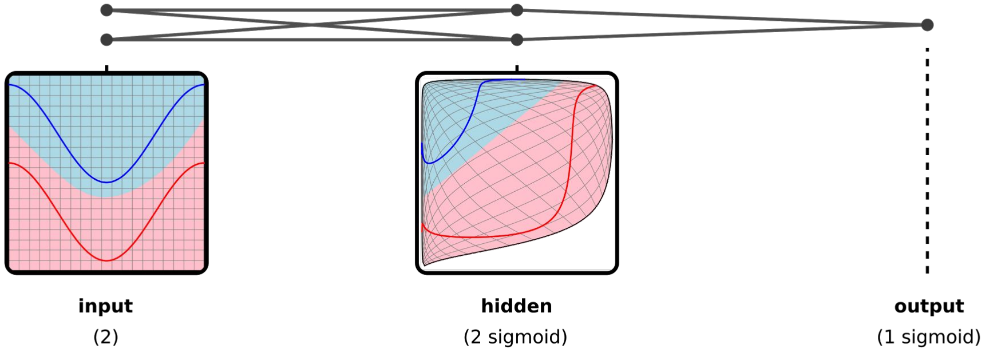


Deep Learning Fundamentals

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.

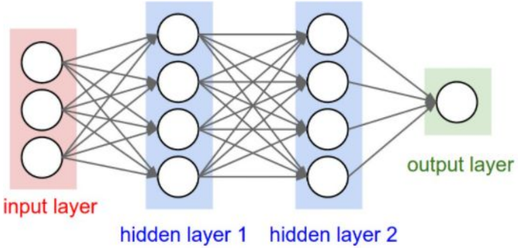


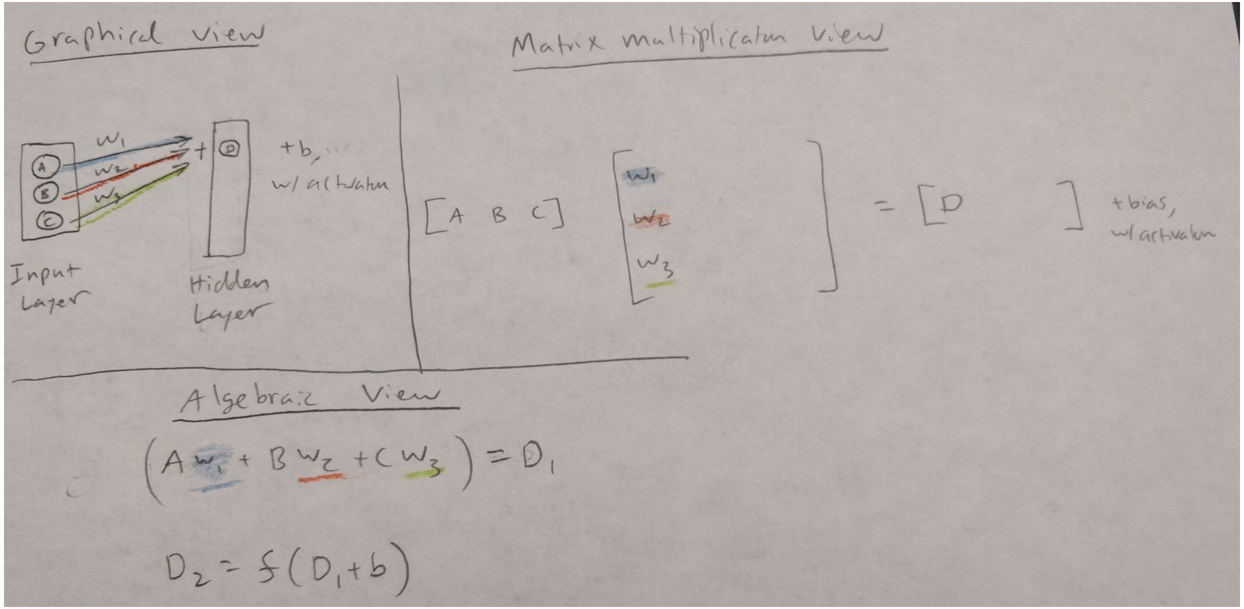


★

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 set 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.




A
B
C
D
w_1
w_2
w_3
+
Input Layer
Hidden Layer

Graphical view

+b, w/ activation

Matrix multiplication view

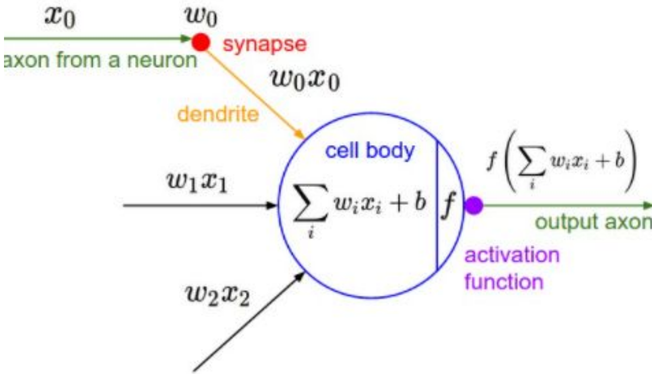
$$[A\ \ B\ \ C]\begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} = [D]$$

+bias, w/ activation

Algebraic View

$$(Aw_1 + Bw_2 + Cw_3) = D_1$$

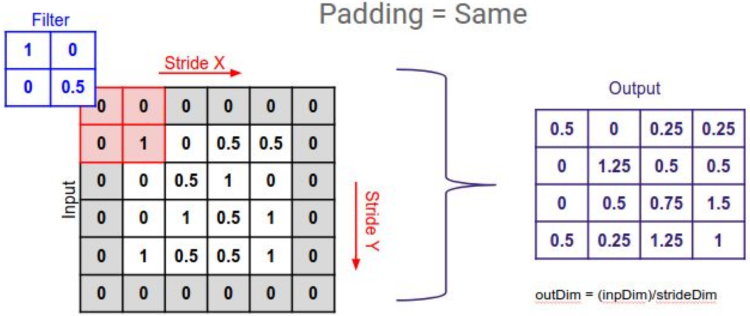
$$D_2 = f(D_1 + b)$$



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**

Padding = Same



Filter

1	0
0	0.5

Input

0	0	0	0	0	0
0	1	0	0.5	0.5	0
0	0	0.5	1	0	0
0	0	1	0.5	1	0
0	1	0.5	0.5	1	0
0	0	0	0	0	0

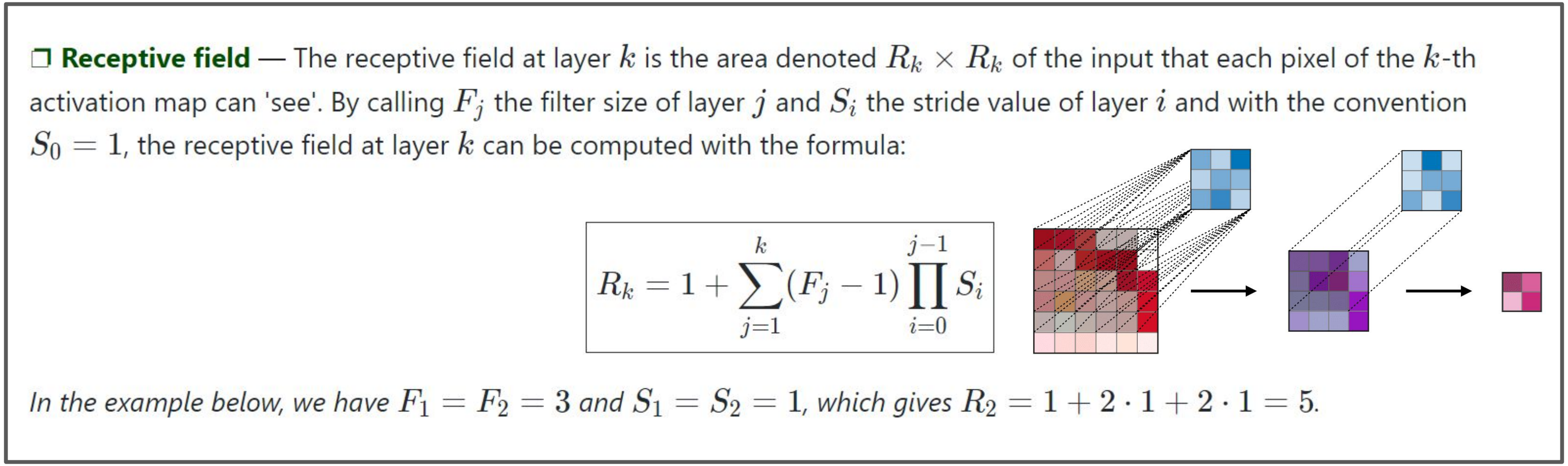
Output

0.5	0	0.25	0.25
0	1.25	0.5	0.5
0	0.5	0.75	1.5
0.5	0.25	1.25	1

outDim = (inpDim)/strideDim

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.



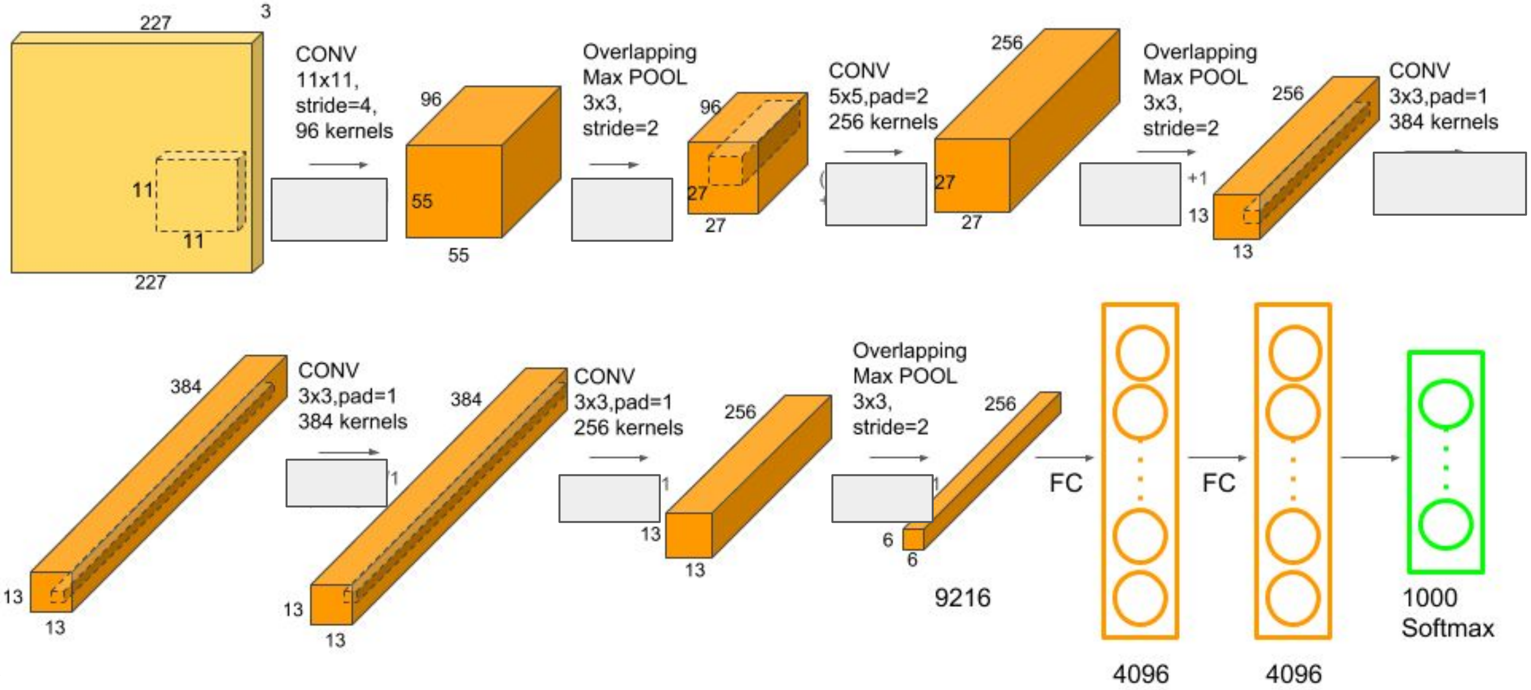
❒ **Receptive field** — The receptive field at layer k is the area denoted $R_k \times R_k$ of the input that each pixel of the k-th activation map can 'see'. By calling F_j the filter size of layer j and S_i the stride value of layer i and with the convention $S_0 = 1$, the receptive field at layer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

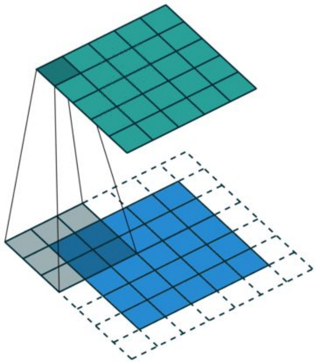


★

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

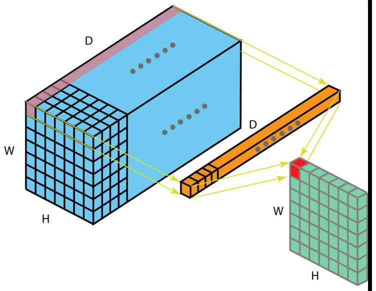
Regular Convolutions

- 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



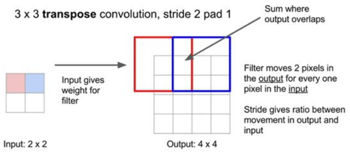
1x1 Convolutions

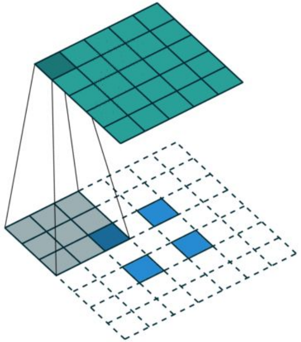
- Same as regular conv, but 1 x 1 x c filters. This keeps the spatial dimensions, but changes the channels
- Useful for dim. reduction, and introducing nonlinearities
- Output array from each filter after convolving with input can be thought of as a weighted sum of the input feature map's channels



De/Up/Transpose/fractionally strided Conv.

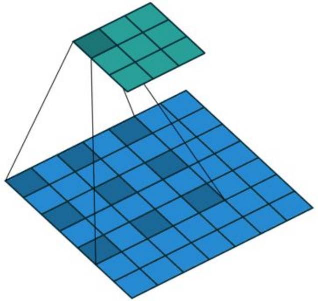
- This applies some fancy padding to achieve an upsampling effect.


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



Atrous/Dilated Conv.

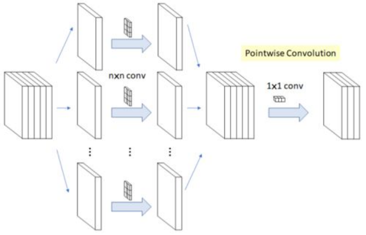
- Adds gaps in the filter when convolving, increasing the field of view even with the same number of parameters.
- However, this does not spatially upsample

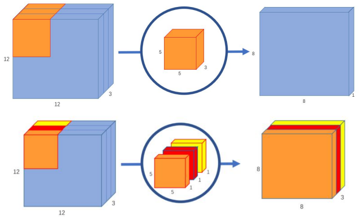


Depthwise Separable Conv. (From MobileNet)

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.

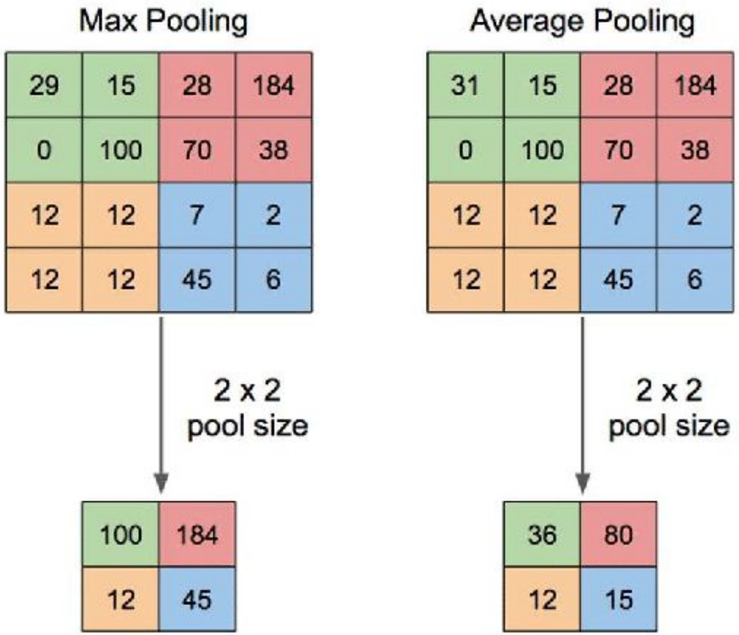




Normal (top) vs depthwise (bottom) conv

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.



Max Pooling

29	15	28	184
0	100	70	38
12	12	7	2
12	12	45	6

100	184
12	45

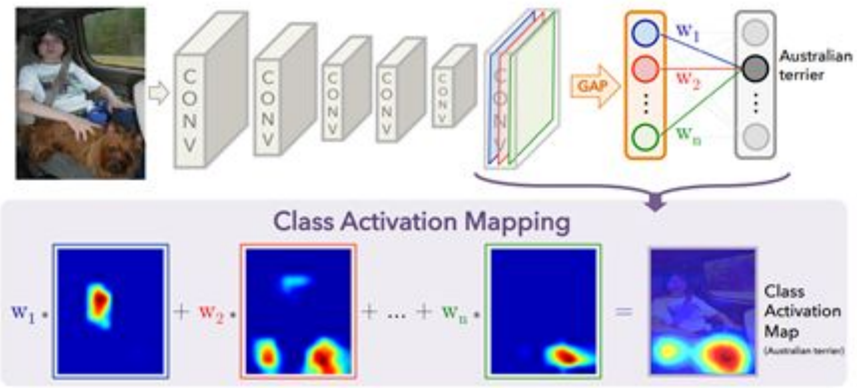
Average Pooling

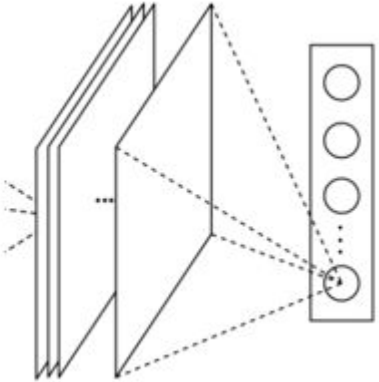
31	15	28	184
0	100	70	38
12	12	7	2
12	12	45	6

36	80
12	15

What is global average pooling?

- First introduced in the Network-In-Network paper, this is designed to replace the last FC layers before softmax classification.
- **One feature map is generated for each class;** then, **each feature map is averaged**, and the resulting **vector is fed directly into the softmax**.
- The global average pooling and softmax has **no parameters**, which is an advantage over FC layers (who are prone to overfitting). It can also be more native to the convolution structure.
  - This is in contrast to standard models, which flatten the last feature maps before feeding it into FC layers, destroying the correspondences.
  - Thus the feature maps can be interpreted as a structural regularizer which causes feature maps to be category confidence maps.
- For example, a tensor with dimensions h×w×d is reduced in size to have dimensions 1×1×d.
- This was used in the Class Activation Mapping paper to visualize results
  - Uses global average pooling (GAP) and weights from a softmax layer to create a class activation map (CAM). The weights corresponding to a class from a softmax layer are used to weigh the GAP feature maps; the sum of them, and upscaling to the image size, creates the CAM.





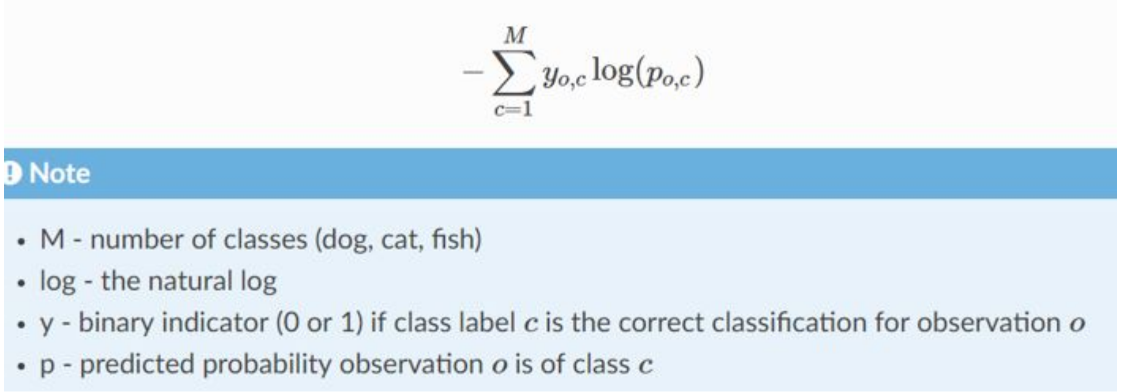
What is the softmax function, and what loss function is it usually associated with?

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

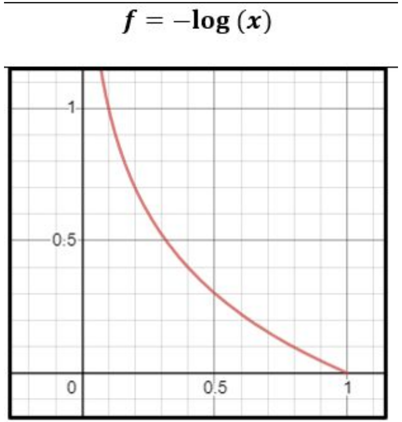
The softmax is often paired with the **cross-entropy loss** (aka **negative log likelihood**), which operates on vectors of class probabilities.



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

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



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

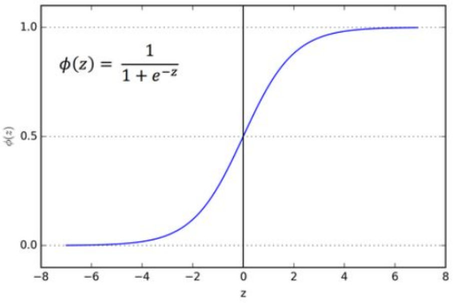


Fig: Sigmoid Function

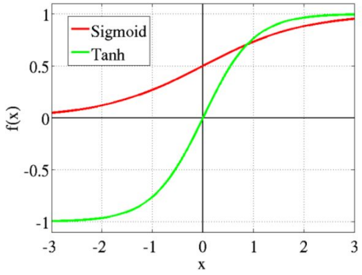
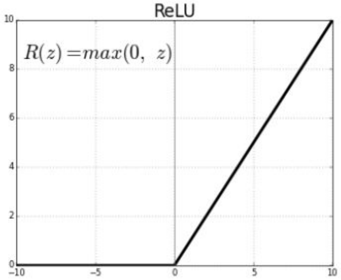
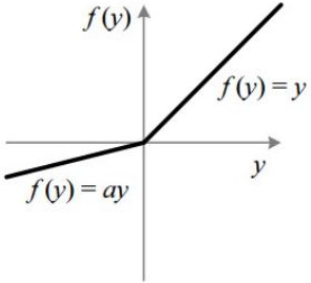


Fig: tanh v/s Logistic Sigmoid





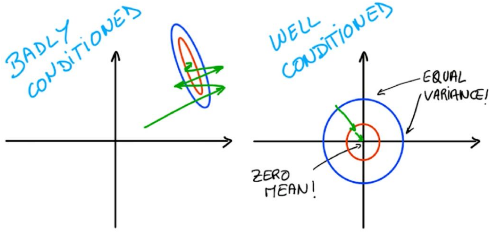
Leaky ReLU (usually a is around 0.01)

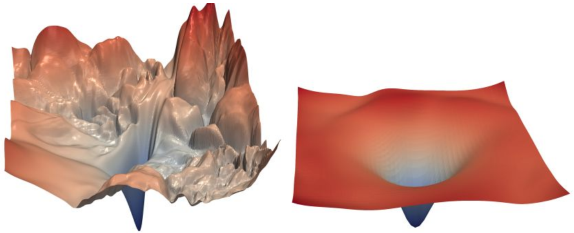
Some notes:

- 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 set it as 0, 1, or 0.5.
- Leaky ReLU avoids the issue of neurons being deactivated if negative.

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.

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

- **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)

★

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**.

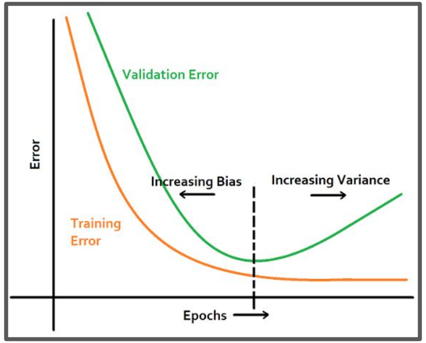
Describe some basic regularization techniques (5).

- **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

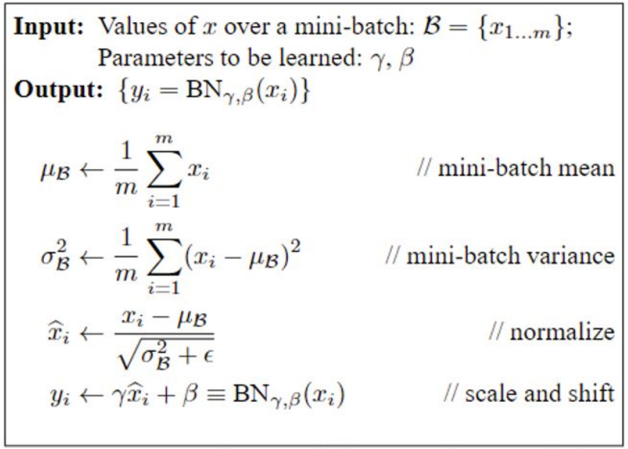


★

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

Implementation Notes:

- Makes training **faster and more stable**
- **For convolutional layers**: we additionally want different elements of the same feature map to be normalized in the same way. Thus, the normalization is performed per feature map, rather than per activation. For example, for a feature map of size p×q and a mini-batch of size m, m×p×q parameters are normalized in the same way, and there is a single γ, β for these m×p×q parameters. This occurs for both training and inference.
- 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 bach 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.



Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

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

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

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

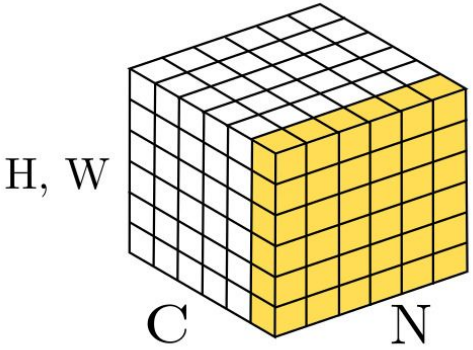
$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Several theories on why batchnorm works:

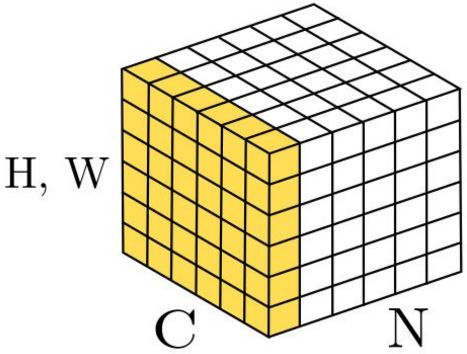
- **Internal Covariate Shift** for a non-batch normalized network slows down training; by forcing distribution of activations to be uniform, there is less adapting necessary.
    - Inputs to each layer are affected by the parameters of all preceding layers – so that small changes to the network parameters 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

What is LayerNorm and how is it different from batchnorm? What are its advantages?

- 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, LayerNorm calculates the statistics per-minibatch sample across all its channels, instead of per-channel across all minibatches



Batch Normalization



Layer Normalization

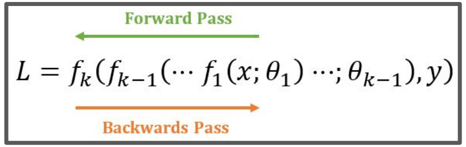
How does backpropagation work in neural networks?

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

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.

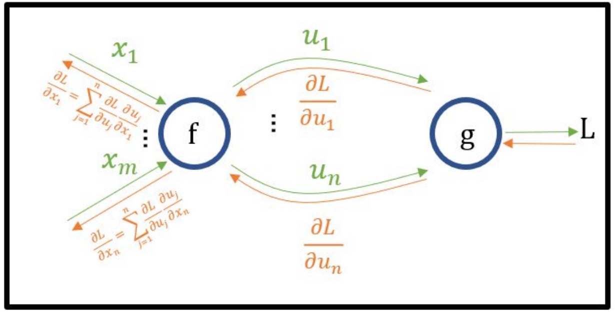


Forward Pass

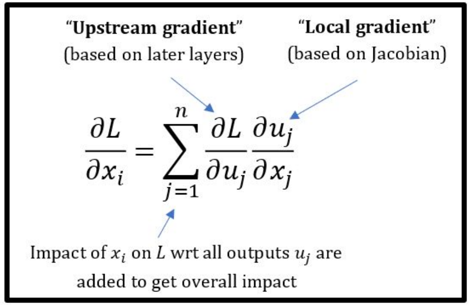
$$L = f_k(f_{k-1}(\cdots f_1(x;\theta_1)\cdots;\theta_{k-1}),y)$$

Backwards Pass

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



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

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

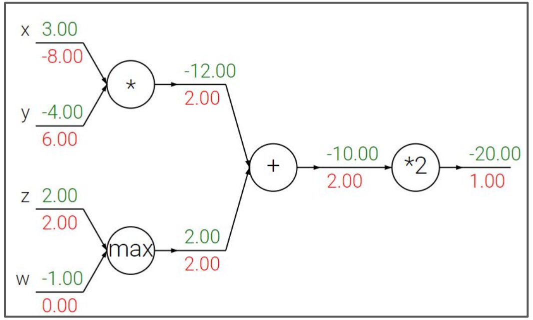
Impact of x_i on L wrt all outputs u_j are added to get overall impact

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)

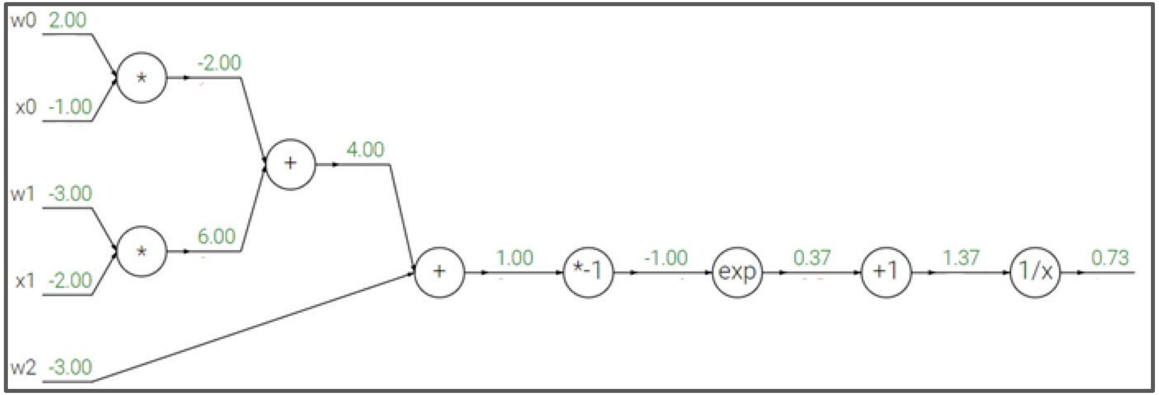
Describe how the following functions/gates propagate the upstream gradient during backpropagation: add, max, multiply

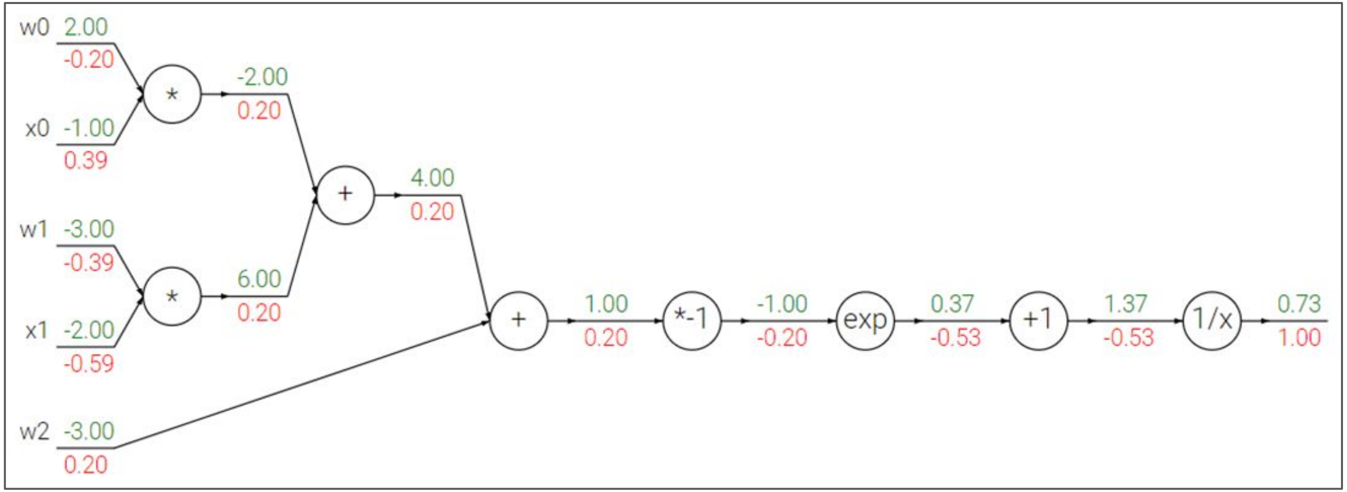
Gate	Behavior	Intuition
Add	Adds all the upstream gradient on its output and distributes that sum equally to all of its inputs. This follows from the fact that the local gradient for the add operation (e.g. x+7) is simply +1.0	Forwards gradients, unchanged.
Max	Distributes the gradient (unchanged) to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

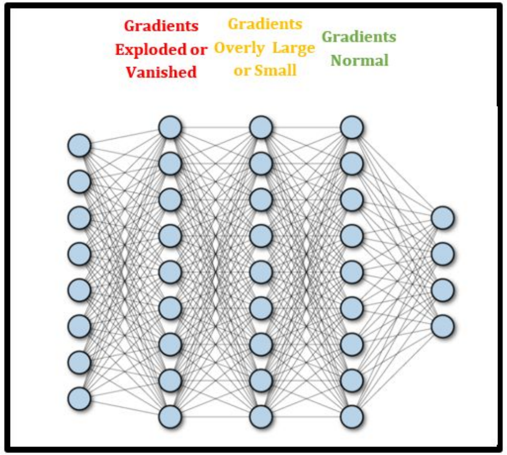
$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$

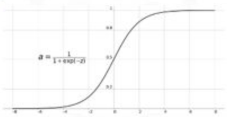
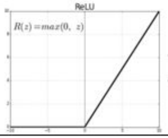
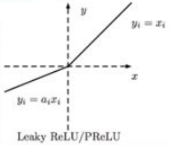




What is the vanishing/exploding gradients problem?

- 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
  - Proper weight initialization
  - BatchNorm


Gradients Exploded or Vanished
Gradients Overly Large or Small
Gradients Normal

	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. Sigmoid Function  $a = \frac{1}{1+\exp(-z)}$  ReLU $R(z) = max(0, z)$  y $y_i = x_i$ x $y_i = a_i x_i$ Leaky ReLU/PReLU	• 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.

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
    - Initializing batchnorm params as 0

At a high level, what are 3 approaches for ML if you only have a small dataset of labeled data and a lot of unlabeled data?

- **Pre-training + Fine Tuning**
  - Only use your labeled data
  - Most straightforward approach
  - Pretraining dataset can be completely different from goal task. Just want to learn good representations.
  - Depending on nature of labeled dataset, may want to apply long-tailed techniques like resampling, reweighing, or margin losses
  - Would want to apply heavy data augmentation and mixup; consistency training (such as making sure augmented have similar responses)
- **Semi-Supervised Learning**
  - Learn from both your labeled and unlabeled data
  - Example approaches:
    - **UDA** on unlabeled + finetuning on labeled, if there’s another similar dataset for the same task, but different domain
    - **Consistency regularization**, that can be applied without labels such on transformations or mixup, to learn general smoothness, clustering, and low-density separation characteristics, in addition to normal supervised learning
    - **Iterative self-training** on the most confident **psuedolabels**
  - Above approaches can be combined
- **Active learning**
  - Find which unlabeled data are the most valuable data to label, given limited budget

★

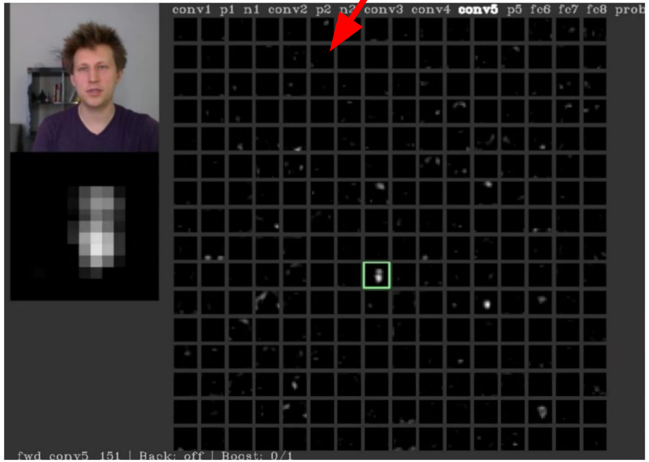
What are some sources of randomness/stochasticity in neural networks? Why is this sometimes desired?

- Randomly **initializing weights/biases**
  - Useful for **ensembles**, 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
- Additionally, when there are symmetries in the action space (like a game of go, or image/music/text generation), randomness helps enable the network to output **multiple possibilities** (possibly in a probabilistic sense)

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
fwd conv5 151 | Back: off | Boost: 0/1

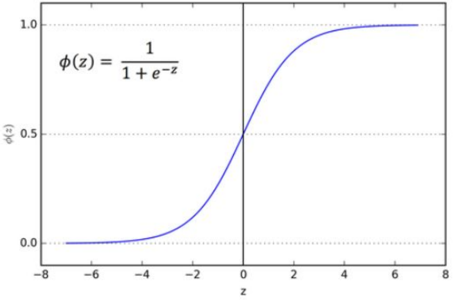


Fig: Sigmoid Function

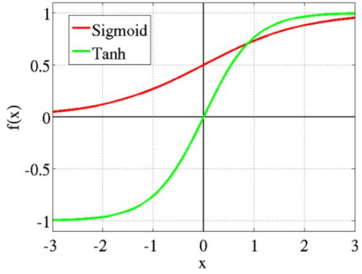
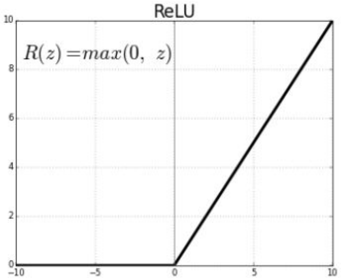
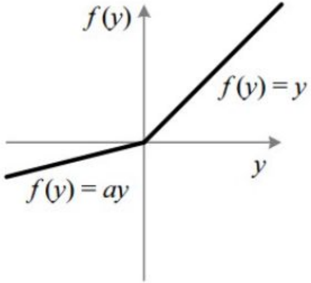


Fig: tanh v/s Logistic Sigmoid

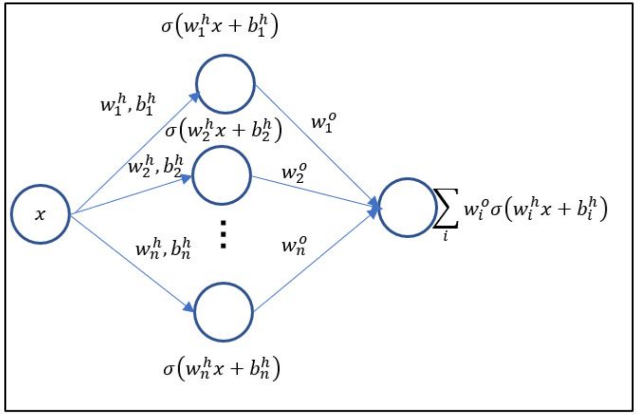




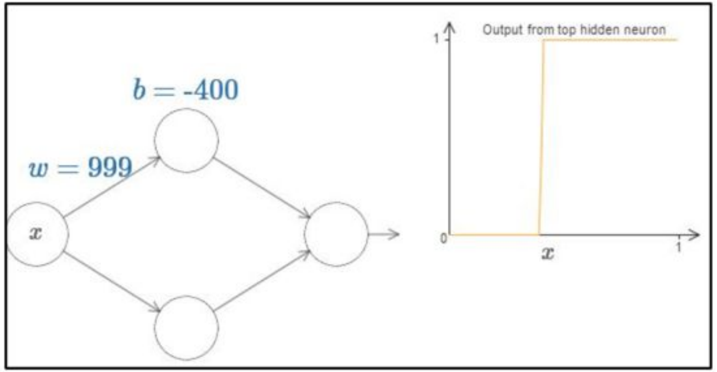
Leaky ReLU (usually a is around 0.01)

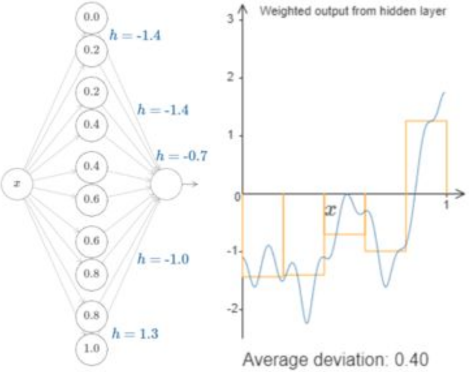
What is the universal approximation theorem, and what are its limitations for practical applications?

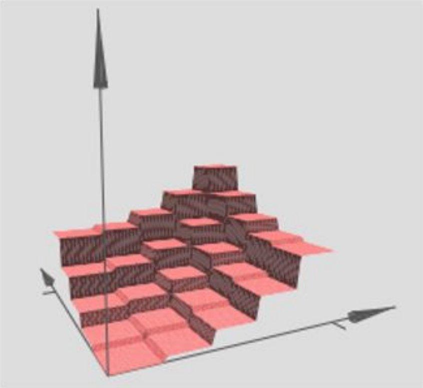
- A striking result that states that a FC network with only one hidden layer can approximate any function to arbitrary precision, given enough width (hidden neurons)
- At a high level, this is because sigmoid can represent step functions, and when enough are combined, can approximate any function
- In 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
$\sigma(w_n^h x + b_n^h)$

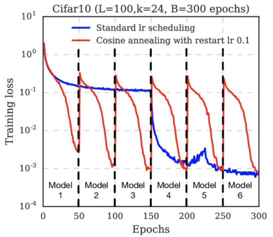


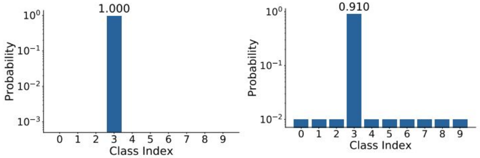




At a high level, summarize the following tricks: Cosine Learning Rate, Label smoothing

- **Cosine learning rate**
  - Method of adjusting learning rate better for SGD
  - Starting with a large learning rate that is relatively rapidly decreased to a minimum value before being increased rapidly again
- **Label smoothing**
  - Intuitively, tries to reduce overconfident predictions and improve **calibration**
  - Accounts for the fact that datasets may have mistakes in them



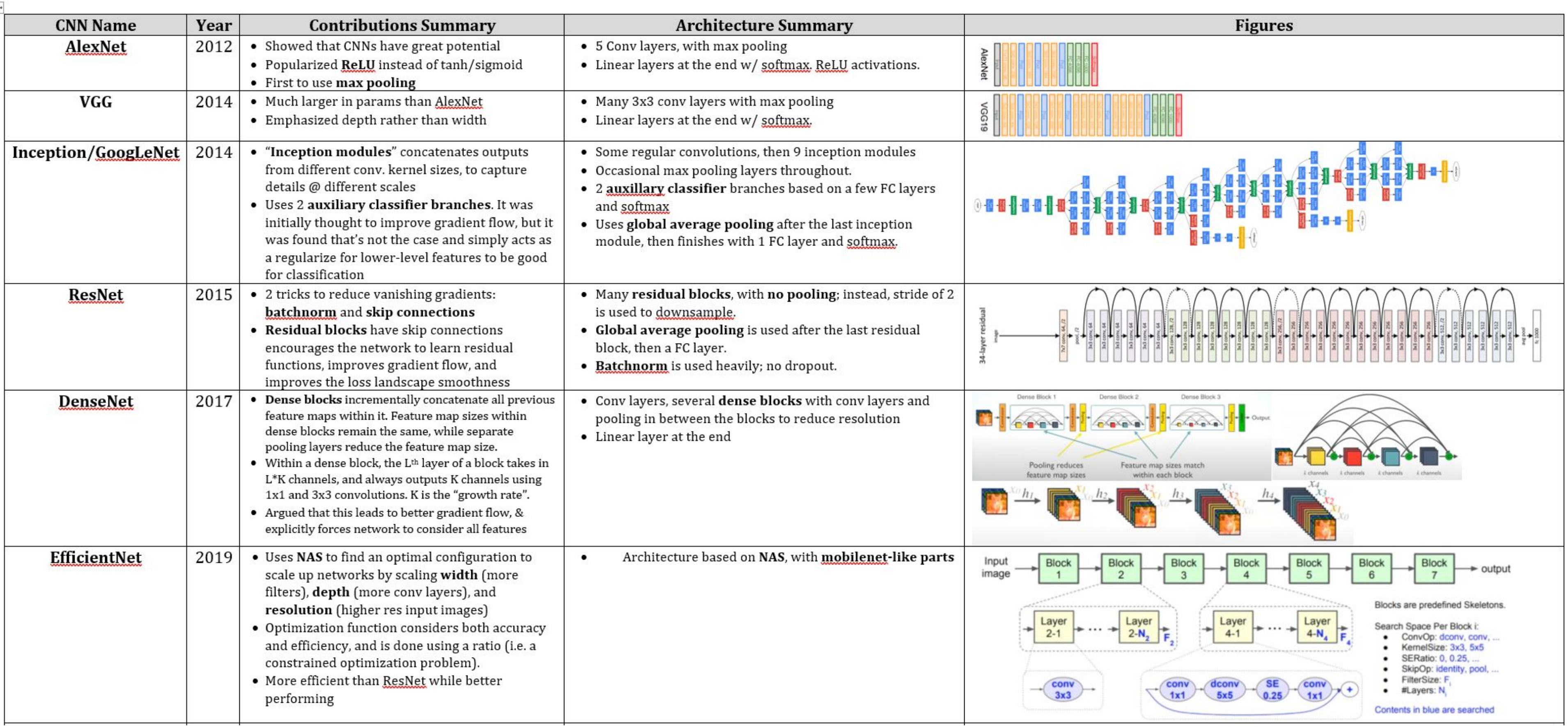


(a) Hard Label

(b) LS

Seminal & Foundational Topics in Deep Learning

At a high level, describe the following CNN backbone architectures: AlexNet, VGG, Inception/GoogLeNet, ResNet, DenseNet, EfficientNet.



CNN Name	Year	Contributions Summary	Architecture Summary	Figures
AlexNet	2012	• Showed that CNNs have great potential • Popularized **ReLU** instead of tanh/sigmoid • First to use **max pooling**	• 5 Conv layers, with max pooling • Linear layers at the end w/ softmax. ReLU activations.	
VGG	2014	• Much larger in params than AlexNet • Emphasized depth rather than width	• Many 3x3 conv layers with max pooling • Linear layers at the end w/ softmax.	
Inception/GoogLeNet	2014	• "**Inception modules**" concatenates outputs from different conv. kernel sizes, to capture details @ different scales • Uses 2 **auxiliary classifier branches**. It was initially thought to improve gradient flow, but it was found that's not the case and simply acts as a regularize for lower-level features to be good for classification	• Some regular convolutions, then 9 inception modules • Occasional max pooling layers throughout. • 2 **auxillary classifier** branches based on a few FC layers and softmax • Uses **global average pooling** after the last inception module, then finishes with 1 FC layer and softmax.	
ResNet	2015	• 2 tricks to reduce vanishing gradients: **batchnorm** and **skip connections** • **Residual blocks** have skip connections encourages the network to learn residual functions, improves gradient flow, and improves the loss landscape smoothness	• Many **residual blocks**, with **no pooling**; instead, stride of 2 is used to downsample. • **Global average pooling** is used after the last residual block, then a FC layer. • **Batchnorm** is used heavily; no dropout.	
DenseNet	2017	• **Dense blocks** incrementally concatenate all previous feature maps within it. Feature map sizes within dense blocks remain the same, while separate pooling layers reduce the feature map size. • Within a dense block, the L^th^ layer of a block takes in L*K channels, and always outputs K channels using 1x1 and 3x3 convolutions. K is the "growth rate". • Argued that this leads to better gradient flow, & explicitly forces network to consider all features	• Conv layers, several **dense blocks** with conv layers and pooling in between the blocks to reduce resolution • Linear layer at the end	Dense Block 1 Dense Block 2 Dense Block 3 Output Pooling reduces feature map sizes Feature map sizes match within each block k channels k channels k channels k channels x_0 h_1 x_1 h_2 x_2 h_3 x_3 h_4 x_4
EfficientNet	2019	• Uses **NAS** to find an optimal configuration to scale up networks by scaling **width** (more filters), **depth** (more conv layers), and **resolution** (higher res input images) • Optimization function considers both accuracy and efficiency, and is done using a ratio (i.e. a constrained optimization problem). • More efficient than ResNet while better performing	• Architecture based on **NAS**, with **mobilenet-like parts**	Input image Block 1 Block 2 Block 3 Block 4 Block 5 Block 6 Block 7 output Layer 2-1 Layer 2-N_2 F_2 Layer 4-1 Layer 4-N_4 F_4 conv 3x3 conv 1x1 dconv 5x5 SE 0.25 conv 1x1 + Blocks are predefined Skeletons. Search Space Per Block i: • ConvOp: dconv, conv, ... • KernelSize: 3x3, 5x5 • SERatio: 0, 0.25, ... • SkipOp: identity, pool, ... • FilterSize: F_i • #Layers: N_i Contents in blue are searched

Additional Notes:

- Resnet also improves the loss landscape, and the addition forwards the gradients backwards, reducing vanishing gradients

★

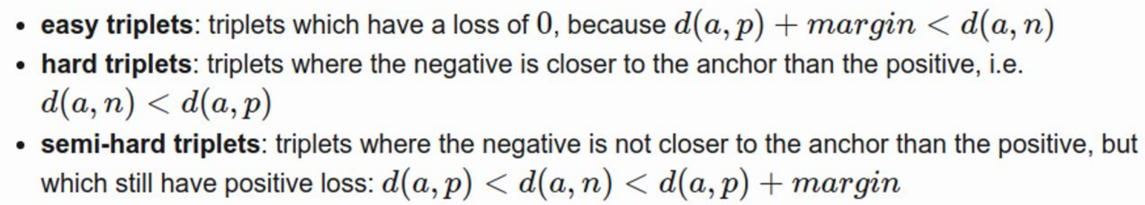
What is the triplet loss, and when is it used? How can it be trained effectively?

- **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

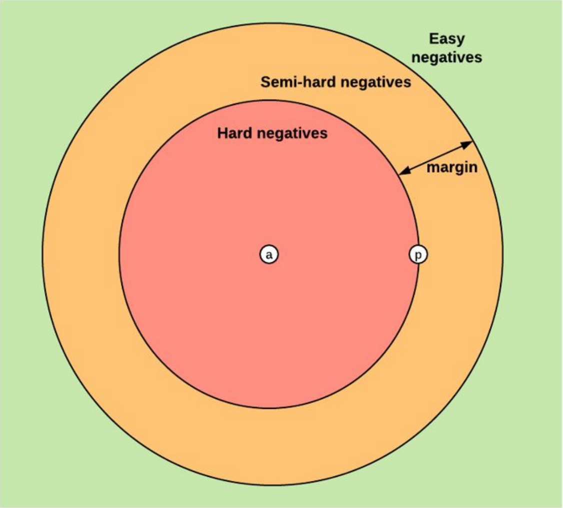
$$\mathcal{L}(A, P, N) = \max\left(\| \mathrm{f}(A) - \mathrm{f}(P)\|^2 - \| \mathrm{f}(A) - \mathrm{f}(N)\|^2 + \alpha, 0\right)$$

- 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:

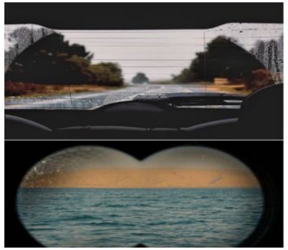


- **easy triplets**: triplets which have a loss of 0, because $d(a,p) + margin < d(a,n)$
- **hard triplets**: triplets where the negative is closer to the anchor than the positive, i.e. $d(a,n) < d(a,p)$
- **semi-hard triplets**: triplets where the negative is not closer to the anchor than the positive, but which still have positive loss: $d(a,p) < d(a,n) < d(a,p) + margin$



At a high level, explain what “Attention”, and “Self-Attention” is. What is soft vs hard attention?

- In general, various **attention mechanisms** allow a network to somehow **weigh features by level of importance** to a task, **and use this weighting to focus on what signals matter, to help achieve the task**.
    - Unlike post-hoc explainability mechanisms like grad-cam, attention can also provide improved performance (on-the-fly).
- It was originally popular in NLP, where attention is necessary to “remember” long-range dependencies in text. However, it can also provide increased explainability and performance to vision tasks, allowing for spatial dependency modeling beyond the receptive field, which can be limited.
- **Soft Attention vs Hard Attention**
    - **Soft Attention**: Can see entire image, but focuses on attended parts
        - Requires more memory/computation but is differentiable.
    - **Hard Attention**: Can only see an attended subset of the image.
        - Requires less computation/memory, but nondifferentiable, so need things like reinforcement learning
- **“Attention” vs “Self-Attention”**
    - **Attention**:
        - Looks to the activations or states of another layer
        - Often used to transfer information, e.g. from encoder to decoder
        - Usually only applied once, to connect 2 components
    - **Self-Attention**:
        - Looks to the activations or states of the same layer where it’s applied
        - Doesn’t connect 2 components; applied within one component
        - Can be applied many times independently in a model



Soft Attention (top, foggy pane of glass) vs hard attention (bottom, binoculars)

How do Non-Local Neural Networks work?

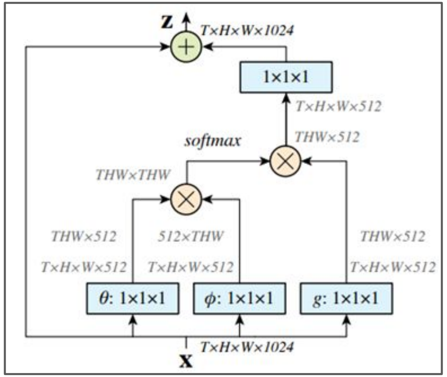
- Reformulates convolutions to make it more global and capable of long-range dependency modeling, instead of relying on local receptive fields (a type of self-attention)
- Found to be useful in both video data and object detection/segmentation (applied to mask-RCNN by adding an additional non-local block at end)
- Also found to be useful in SA-GANs and BigGANs to leverage complementary features from distant portions when generating

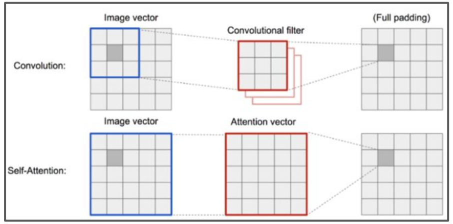
$$\mathbf{y}_i = \frac{1}{\mathcal{C}(\mathbf{x})} \sum_{\forall j} f(\mathbf{x}_i, \mathbf{x}_j) g(\mathbf{x}_j) \quad \mathbf{z}_i = W_z \mathbf{y}_i + \mathbf{x}_i$$

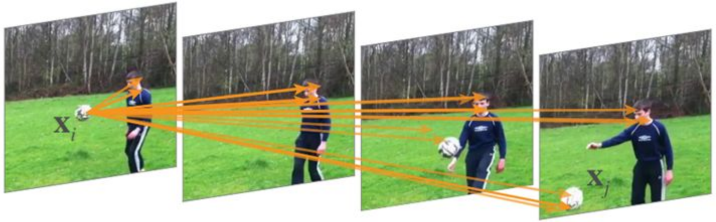
- x is input, f is a pairwise self-attention function to model relationships, g modifies the input signal, and j is an index which enumerates all possible positions. $C(x)$ is a normalization factor
- y is intermediate output, z is the final output, which considers all locations in the input
- Example instantiation:

  $f(\mathbf{x}_i, \mathbf{x}_j) = e^{\theta(\mathbf{x}_i)^T \phi(\mathbf{x}_j)}$

  $g(x_j) = W_g x_j$







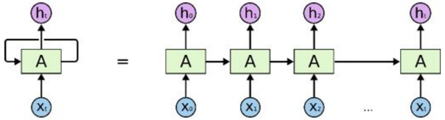
SA-GAN

Neural Networks Designed for Sequential Data

(RNNs, LSTMs, Transformers)

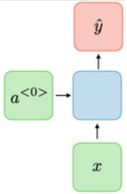
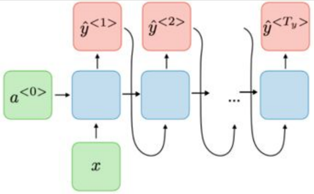
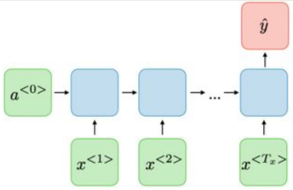
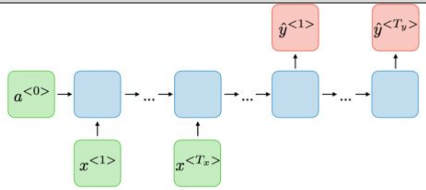
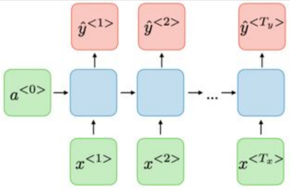
What can RNNs be used for, and how do they work?

- **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.
- 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.

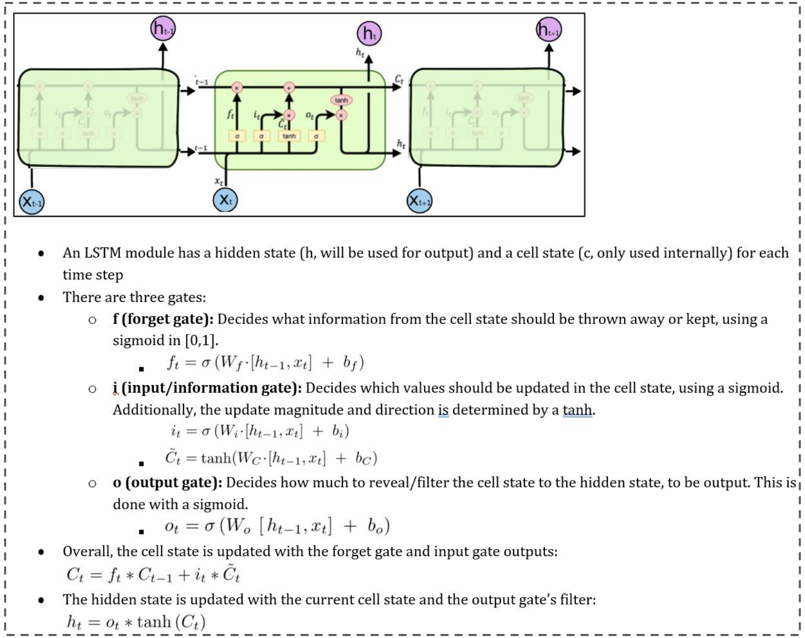
- In general, $h_t = f_w(h_{t-1}, x_t) = hidden\ state\ at\ time\ t$
  - $f_w = some\ function\ with\ parameters\ w$
    - Weights and function are fixed – don't change with time
  - $h_{t-1} = previous\ time\ step's\ hidden\ state$
  - $x_t = input\ @\ time\ t$
- In general, $y_t = f_y(h_t) = output\ for\ time\ t$
- 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		 $\hat{y}^{<1>}$ $\hat{y}^{<2>}$ $\hat{y}^{<T_y>}$ $a^{<0>}$ ... x	 $\hat{y}$ $a^{<0>}$ ... $x^{<1>}$ $x^{<2>}$ $x^{<T_x>}$	 $\hat{y}^{<1>}$ $\hat{y}^{<T_y>}$ $a^{<0>}$ ... ... $x^{<1>}$ $x^{<T_x>}$	 $\hat{y}^{<1>}$ $\hat{y}^{<2>}$ $\hat{y}^{<T_y>}$ $a^{<0>}$ ... $x^{<1>}$ $x^{<2>}$ $x^{<T_x>}$
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

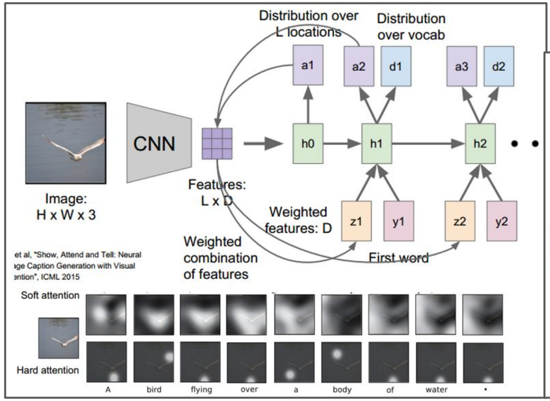
★

What are LSTMs used for and how do they work?

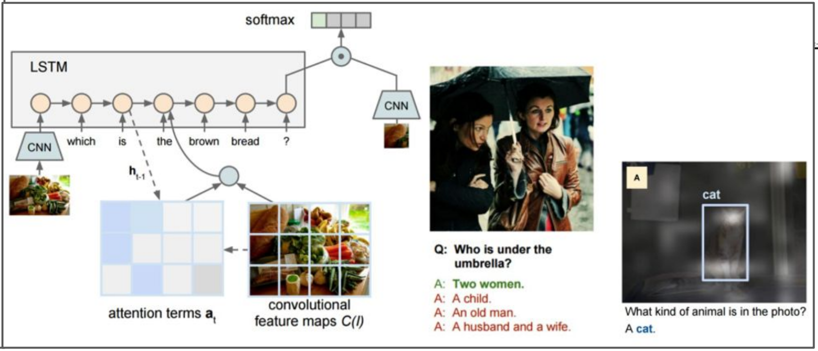
- **Long Short Term Memory (LSTM)** is a RNN variant solves issues with exploding/vanishing gradients by two modifications, which improve gradient flow (at a high level, similar to ResNets' residuals):
  - Incorporate a **cell state** (c) used only internally for each time step. The hidden state (h) 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.
- 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)**


h_{t-1}
h_t
h_{t+1}
C_{t-1}
C_t
h_{t-1}
h_t
f_t
i_t
$\tilde{C}_t$
o_t
σ
σ
tanh
σ
tanh
x_t
X_{t-1}
X_t
X_{t+1}

- 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:
  $h_t = o_t * \tanh\left(C_t\right)$



et al, "Show, Attend and Tell: Neural ...ge Caption Generation with Visual ...ntion", ICML 2015



Q: Who is under the umbrella?

A: Two women.

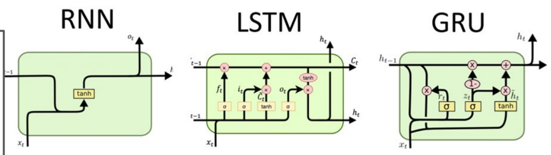
A: A child.

A: An old man.

A: A husband and a wife.

What kind of animal is in the photo?

A cat.



In NLP, what are Transformers and how do they work? 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 **good modeling of long-range dependencies** compared to RNNs/LSTMs
  - Workarounds like **"chunking"** are necessary for inputs larger than the maximum size (though there are approaches for an infinite context, e.g. Transformer-XL)
  - Does not need to process the beginning of the sentence before the end, allowing for **parallelism**
  - Along with the use of skip connections, this partially **addresses the recurrent vanishing gradients** issue from backpropagation through time
- 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
- Due to the removal of hidden states, LSTMs are still common in reinforcement learning
- Two well-known models based on Transformers are:
  - **BERT** is encoder-only and is non-autoregressively. 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:

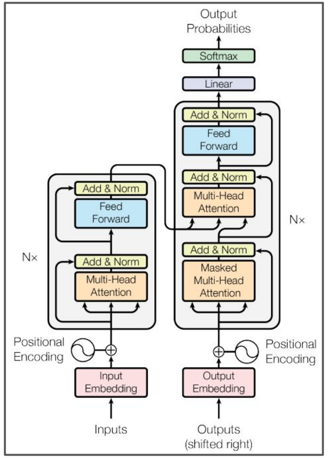
- Input words are embedded into fixed length vectors (e.g. 512 dims)
- Positional encodings are added to each word embedding
- With inputs in matrix X, this is fed into the first encoder block, shown in the figure

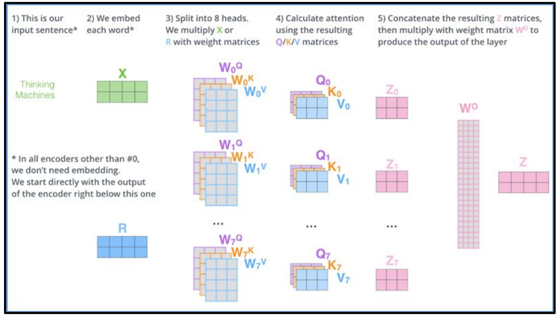
Multi-Head Attention:

- Each head in the **multi-head attention module** contains parameter matrices. E.g. for 8 heads, we would have matrices $W_0^Q, W_0^K, W_0^V, \ldots, W_8^Q, W_8^K, W_8^V$ as well as an overall matrix W^O. The heads are analogous to different filters in a CNN. By using multiple, we hope to increase robustness and capture different unique aspects of the input.
- The usage of Q, K, and V and are inspired by information retrieval systems
  - **Q contains queries:** weights specific for the current word
  - **K contains keys:** weights specific for all words
  - **V contains values:** semantic features
- In the ith head, we calculate $Z_i = softmax\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right)V_i, where\ Q_i = XW_i^Q, K_i = XW_i^K, V_i = XW_i^V$
  - Intuitively, we multiply Q and K, creating pairwise dot products (which are similar to cosine similarities) which are then normalized into weights using a softmax. This represents a **self-attention matrix**: how much a certain word should "focus" on other words.
  - Values are then selected using the attention weights
- Then, all Z_i matrices from each head are concatenated and multiplied with W^O to obtain Z, which is of the same size as X
- Z is added with X (i.e. residual), then normalized. Then, each word is separately fed into the same linear layer with ReLU, with another residual connection, and finally normalized.

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




1) This is our input sentence*

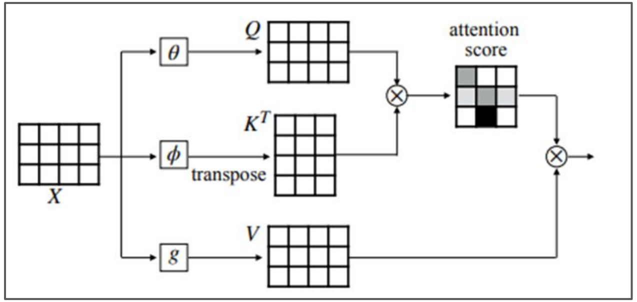
2) We embed each word*

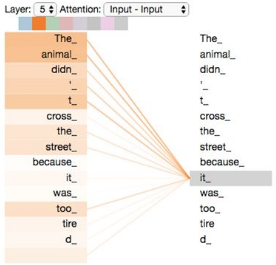
3) Split into 8 heads. We multiply X or R with weight matrices

4) Calculate attention using the resulting Q/K/V matrices

5) Concatenate the resulting Z matrices, then multiply with weight matrix W^O to produce the output of the layer

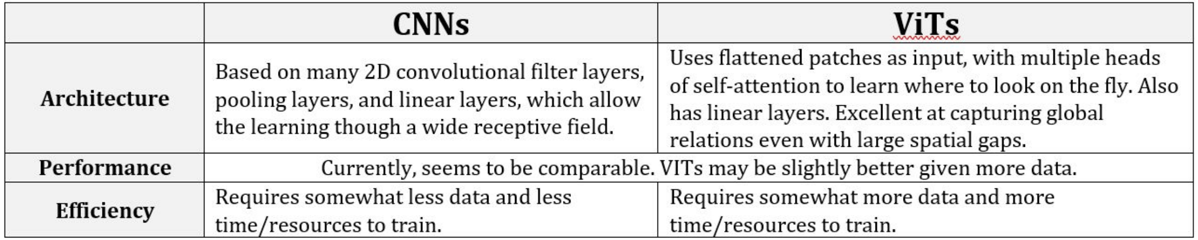
* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one



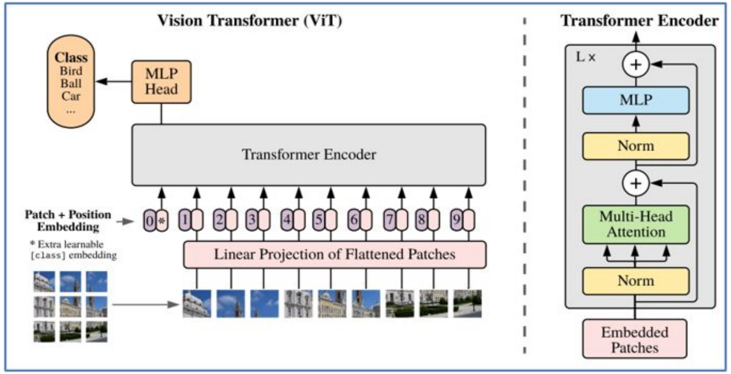

Layer: 5 Attention: Input - Input

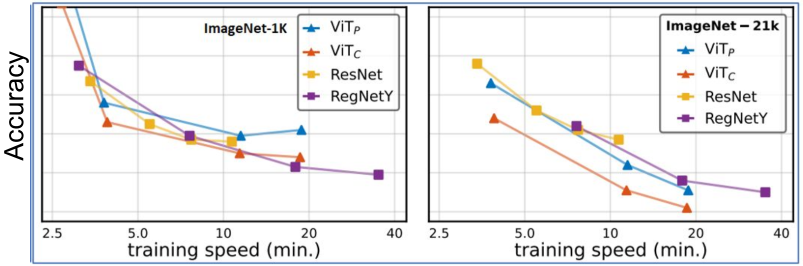
How can Transformers be applied to vision? How do they differ from CNNs?

- ViTs were first introduced in the paper "An image is worth 16x16 words" by Google, ICLR 2021
- They essentially break down input images into patches using convolutions, which are flattened into vectors (i.e. “words”) to be put into a Transformer encoder with multi-head self attention to each patch.
    - Intuitively, repeated weighings of the feature maps, based on cosine similarity of embedded image patches
- Intuitively, this should enable the capture of relationships between different portions of an image.



	CNNs	ViTs
Architecture	Based on many 2D convolutional filter layers, pooling layers, and linear layers, which allow the learning though a wide receptive field.	Uses flattened patches as input, with multiple heads of self-attention to learn where to look on the fly. Also has linear layers. Excellent at capturing global relations even with large spatial gaps.
Performance	Currently, seems to be comparable. VITs may be slightly better given more data.	
Efficiency	Requires somewhat less data and less time/resources to train.	Requires somewhat more data and more time/resources to train.

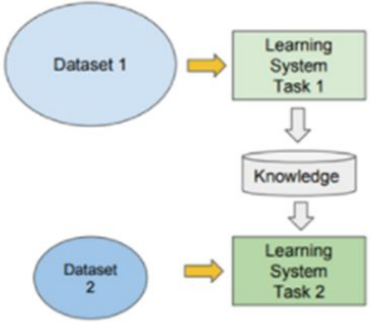




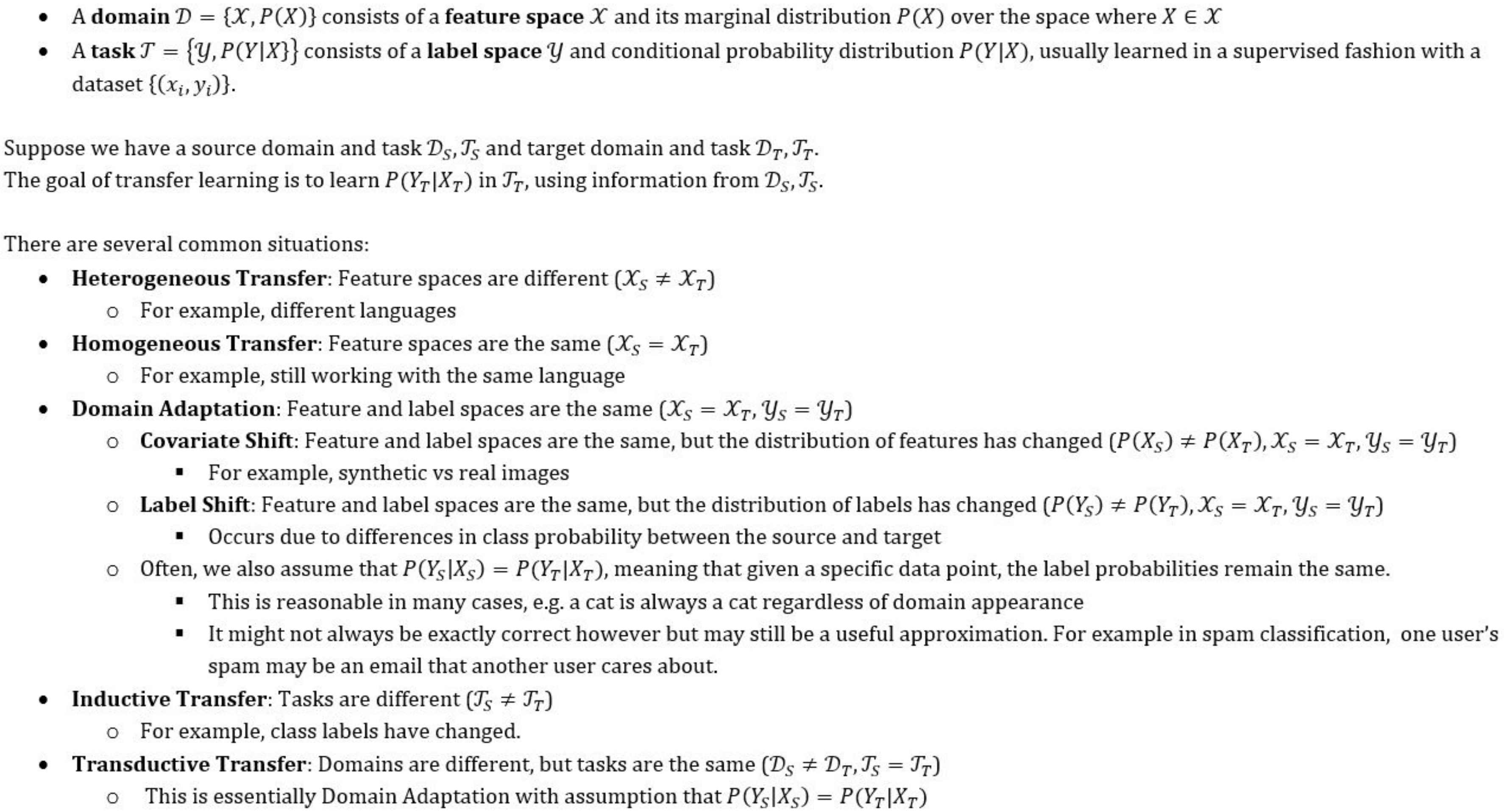
Transfer Learning

What is transfer learning and when is it useful? How is it related to semi-supervised learning and few-shot learning?

- Transfer learning is the application of a model trained on a source task, and applying it to a target task. Different variants can be defined depending on if the inputs between the tasks are different, if the desired outputs are different, or both.
- It should especially be considered if you have a lot of labeled data in the source task, but only a little labeled data in the target task (or, even none at all).
- **Semi-Supervised Learning**
  - Can overlap with transfer learning, in the standard domain adaptation setting (i.e. involves two different tasks where only labels are partially present)
- **Few Shot Learning**
  - Takes transfer learning to the extreme, aiming to learn a target task with only a few, one, or even zero instances of a class.
  - While currently a very challenging problem, it is something that comes naturally to us humans. Toddlers only need to be told once what a dog is in order to be able to identify any other dog (i.e. one-shot learning), while adults can understand the essence of an object just by reading about it in context, without ever having encountered it before (i.e. zero-shot learning).



Describe how to formally define transfer learning, and state some common settings.



- A **domain** $\mathcal{D} = \{\mathcal{X}, P(X)\}$ consists of a **feature space** $\mathcal{X}$ and its marginal distribution $P(X)$ over the space where $X \in \mathcal{X}$
- A **task** $\mathcal{T} = \{\mathcal{Y}, P(Y|X)\}$ consists of a **label space** $\mathcal{Y}$ and conditional probability distribution $P(Y|X)$, usually learned in a supervised fashion with a dataset $\{(x_i, y_i)\}$.

Suppose we have a source domain and task $\mathcal{D}_S, \mathcal{T}_S$ and target domain and task $\mathcal{D}_T, \mathcal{T}_T$.
The goal of transfer learning is to learn $P(Y_T|X_T)$ in $\mathcal{T}_T$, using information from $\mathcal{D}_S, \mathcal{T}_S$.

There are several common situations:

- **Heterogeneous Transfer**: Feature spaces are different ($\mathcal{X}_S \neq \mathcal{X}_T$)
  - For example, different languages
- **Homogeneous Transfer**: Feature spaces are the same ($\mathcal{X}_S = \mathcal{X}_T$)
  - For example, still working with the same language
- **Domain Adaptation**: Feature and label spaces are the same ($\mathcal{X}_S = \mathcal{X}_T, \mathcal{Y}_S = \mathcal{Y}_T$)
  - **Covariate Shift**: Feature and label spaces are the same, but the distribution of features has changed ($P(X_S) \neq P(X_T), \mathcal{X}_S = \mathcal{X}_T, \mathcal{Y}_S = \mathcal{Y}_T$)
    - For example, synthetic vs real images
  - **Label Shift**: Feature and label spaces are the same, but the distribution of labels has changed ($P(Y_S) \neq P(Y_T), \mathcal{X}_S = \mathcal{X}_T, \mathcal{Y}_S = \mathcal{Y}_T$)
    - Occurs due to differences in class probability between the source and target
  - Often, we also assume that $P(Y_S|X_S) = P(Y_T|X_T)$, meaning that given a specific data point, the label probabilities remain the same.
    - This is reasonable in many cases, e.g. a cat is always a cat regardless of domain appearance
    - It might not always be exactly correct however but may still be a useful approximation. For example in spam classification, one user's spam may be an email that another user cares about.
- **Inductive Transfer**: Tasks are different ($\mathcal{T}_S \neq \mathcal{T}_T$)
  - For example, class labels have changed.
- **Transductive Transfer**: Domains are different, but tasks are the same ($\mathcal{D}_S \neq \mathcal{D}_T, \mathcal{T}_S = \mathcal{T}_T$)
  - This is essentially Domain Adaptation with assumption that $P(Y_S|X_S) = P(Y_T|X_T)$

What are some common ways to perform transfer learning, and what are some rules of thumb of when to use each of them?

Labels available in target domain (i.e. “supervised transfer learning”):

- **Pretrained CNN as a fixed feature extractor**
  - Remove the last FC layer, and utilize the feature vectors without changing the weights of the pretrained CNN
  - You can train a linear classifier (e.g. a linear SVM or softmax NN classifier with a few newly added layers) to utilize those features
  - Smaller risk of overfitting but less powerful: in general, better approach if your new dataset is small and/or similar to the original dataset
- **Fine-tuning convnet**
  - Finetune some or all the weights of the pretrained network by continuing the backpropagation
  - Usually, the later layers are finetuned, since they are more domain-specific, while the earlier layers are more generic and low-level and kept static, or **gradually unfrozen**
  - Here, it’s common to use a smaller learning rate
  - Larger risk of overfitting but more powerful: in general, better approach if your dataset is larger and/or different to the original dataset
- **Multi-task Learning**
  - If you have enough computational resources and have access to the original training data, may want to learn the source and target tasks simultaneously
  - Ensures that the source task is not “forgotten”
  - If doing this, make sure that the number of training samples for each task is accounted for, or else you can have imbalance issues
- **Continuous Learning**
  - This is a variant of multi-task learning where we want to enable a model to learn continuously without forgetting.
  - Can be seen as a lifelong transfer learning problem where you don’t forget your source tasks as you try to learn new target tasks.

Labels unavailable in target domain, only available in source domain (i.e. “target unsupervised transfer learning”):

- In this case, normally the the tasks would remain the same (or else, having different tasks and no labels would make the problem too challenging and unconstrained)
- Thus, you generally perform unsupervised domain adaptation here

★

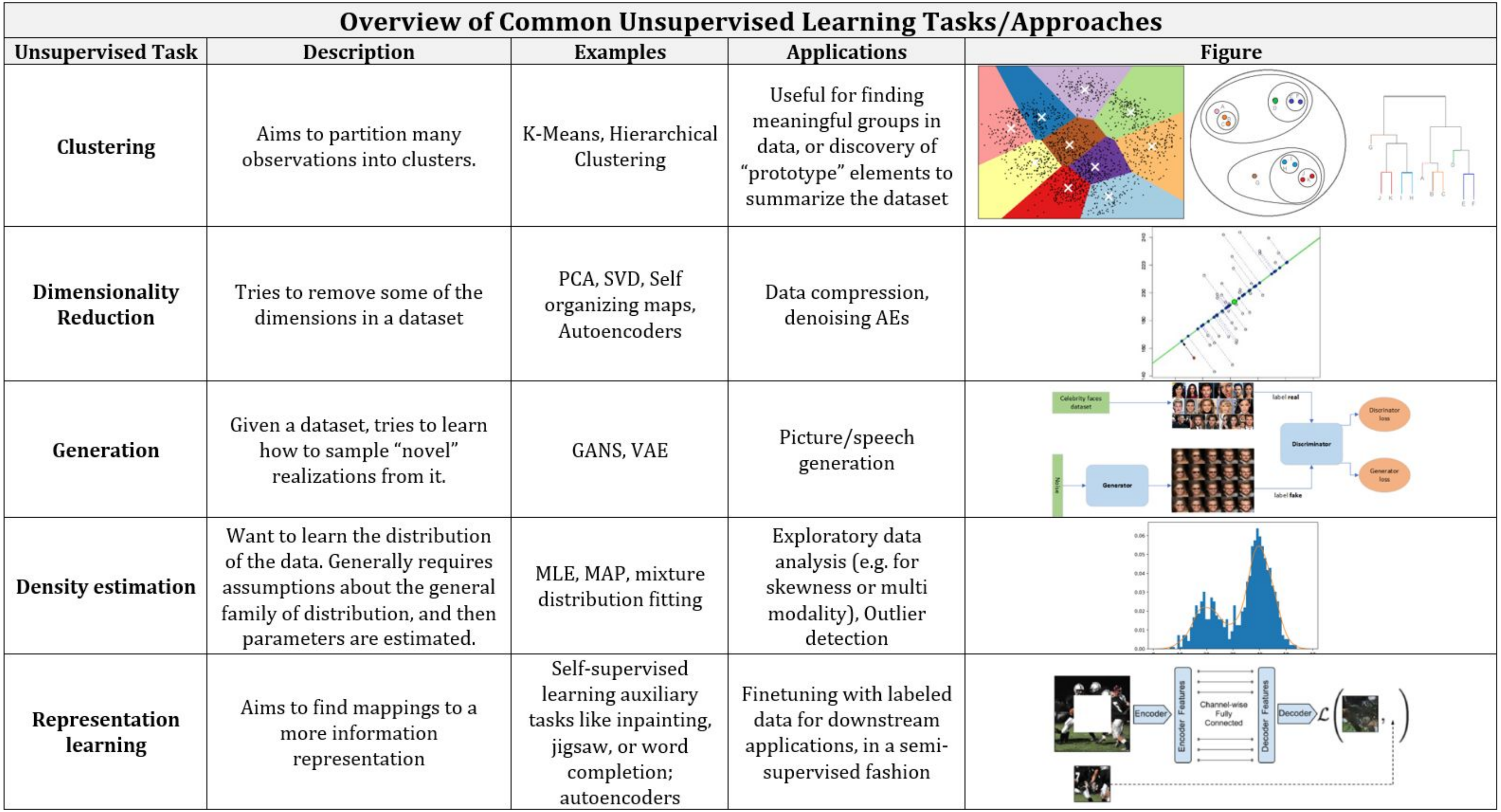
Unsupervised & Self-Supervised Learning

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.



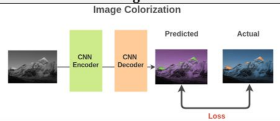
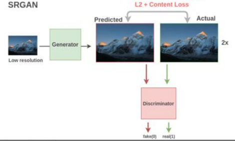
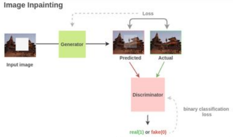
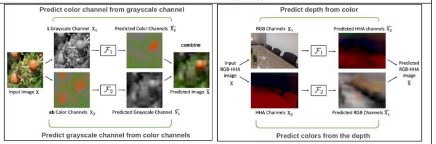
Overview of Common Unsupervised Learning Tasks/Approaches				
Unsupervised Task	**Description**	**Examples**	**Applications**	**Figure**
Clustering	Aims to partition many observations into clusters.	K-Means, Hierarchical Clustering	Useful for finding meaningful groups in data, or discovery of "prototype" 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	

★

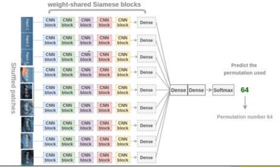
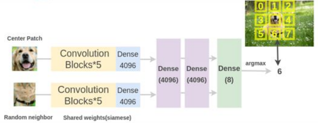
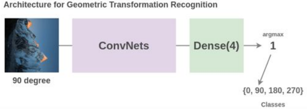
What is self-supervised learning? Explain some common methods based on reconstruction, “common sense”, and automatic labels.

- 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).”

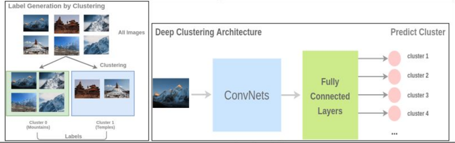
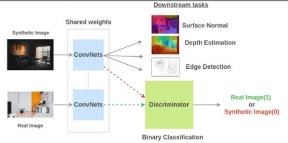
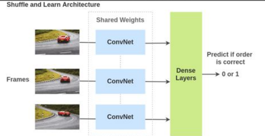
Reconstruction Based SSL

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 Colorization CNN Encoder CNN Decoder Predicted Actual Loss
Image Superresolution	What if we prepared training pairs of (small, upscaled) images by downsampling millions of images we have freely available?	 SRGAN L2 + Content Loss Predicted Actual Low resolution Generator 2x Discriminator fake(0) real(1)
Image Inpainting	What if we prepared training pairs of (corrupted, fixed) images by randomly removing part of images?	 Image Inpainting Loss Input image Generator Predicted Actual Discriminator binary classification loss real(1) or fake(0)
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.	 Predict color channel from grayscale channel L Grayscale Channel X_1 Predicted Color Channels $\widehat{X_2}$ Input Image X $\mathcal{F}_1$ $\mathcal{F}_2$ combine Predicted Image $\widehat{X}$ ab Color Channels X_2 Predicted Grayscale Channel $\widehat{X_1}$ Predict grayscale channel from color channels Predict depth from color RGB Channels X_1 Predicted HHA channels $\widehat{X_2}$ Input RGB-HHA image X Predicted RGB-HHA image $\widehat{X}$ HHA Channels X_2 Predicted RGB Channels $\widehat{X_1}$ Predict colors from the depth

Common-Sense Based SSL

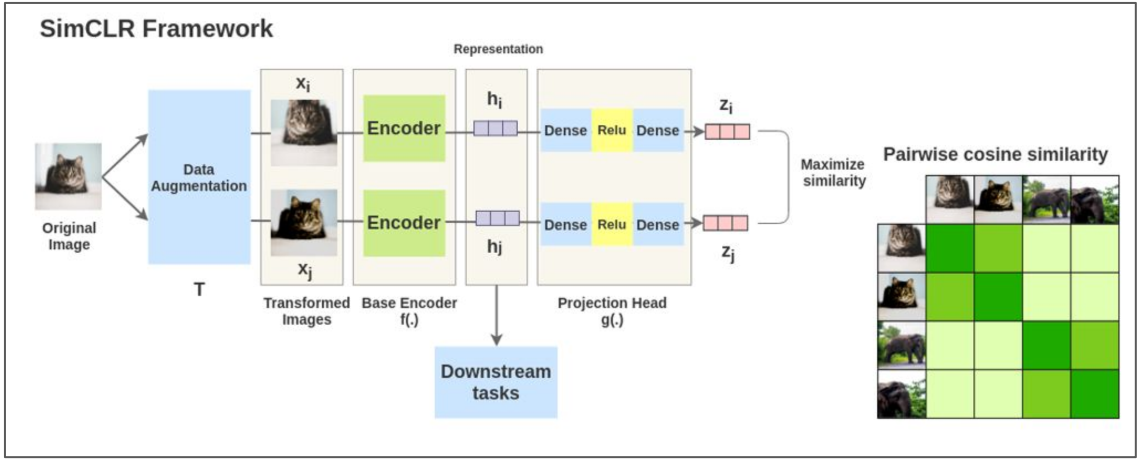
Auxiliary Task Name	Description	Figure
Image Jigsaw Puzzle	What if we prepared training pairs of (shuffled, ordered) puzzles by randomly shuffling patches of images?	 weight-shared Siamese blocks Shuffled patches CNN block Dense Softmax 64 Predict the permutation used Permutation number 64
Context Prediction	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?	 Center Patch Convolution Blocks*5 Dense 4096 Dense (4096) Dense (4096) Dense (8) argmax 6 0 1 2 3 4 5 6 7 Random neighbor Shared weights(siamese)
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?	 Architecture for Geometric Transformation Recognition 90 degree ConvNets Dense(4) argmax 1 {0, 90, 180, 270} Classes

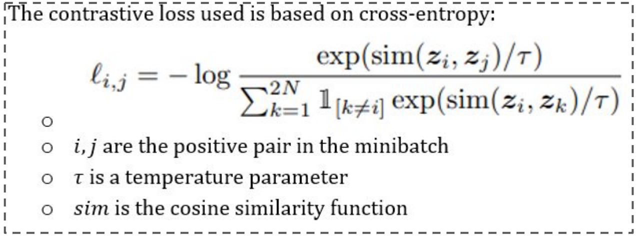
Automatic-Labels Based SSL

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?	 Label Generation by Clustering All Images Clustering Cluster 0 (Mountains) Cluster 1 (Temples) Labels Deep Clustering Architecture Predict Cluster ConvNets Fully Connected Layers cluster 1 cluster 2 cluster 3 cluster 4 ...
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?	 Downstream tasks Shared weights Synthetic Image ConvNets Surface Normal Depth Estimation Edge Detection Real Image Discriminator Real Image(1) or Synthetic Image(0) Binary Classification
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?	 Shuffle and Learn Architecture Shared Weights Frames ConvNet Dense Layers Predict if order is correct 0 or 1

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

Semi-Supervised Learning

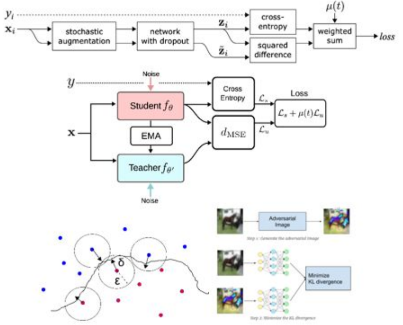
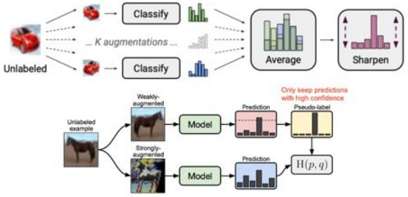
At a high level, what is semi-supervised learning, and what are the common approaches/assumptions?

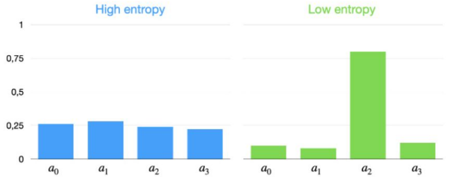
- Goal of semi-supervised learning is to learn with some data that's labeled, and some that is not.
  - Note that regardless of the training method, supervised pretraining on a different dataset/task is still generally helpful.

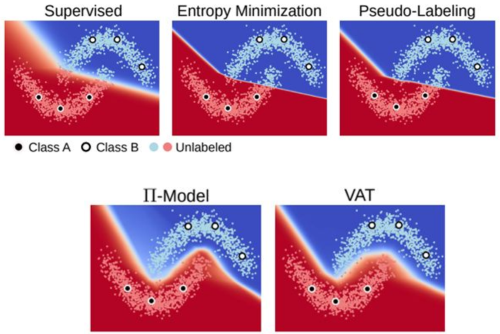
- Several hypotheses have been discussed in literature to support certain design decisions in semi-supervised learning methods:
  - **Smoothness Assumptions:** If two data samples are close in a high-density region of the feature space, their labels should be the same or very similar.
  - **Cluster Assumptions**: 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.

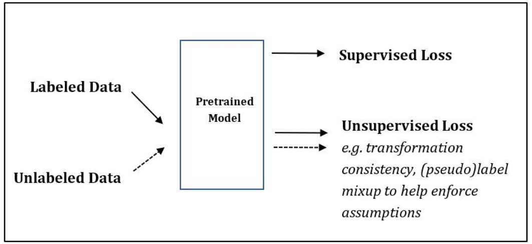
★

Explain the following approaches to semi-supervised learning: GANs, UDA, Consistency regularization, pseudolabeling/self-training.

Overview of Common Semi-Supervised Learning Approaches			
Method	**High-Level Description**	**Examples**	**Figure**
GAN Discriminator	Use a GAN setup to obtain a well-trained discriminator classifier using labeled and unlabeled data.	• Modify discriminator to classify K+1 classes: K labels, and 1 if it is fake. It will be supervised if given labeled training samples; generated samples should be classified as the $K + 1^{th}$ class; unlabeled real examples can be classified as any class except the $K + 1^{th}$. • **GoodBadGAN** argues we don't actually want a good generator (since we already have plenty of real, unlabeled data). Instead, we want a generator that does not match the true data distribution, and serves only to help complement the classification of the labeled examples. It may also expose the classifier to a more diverse set of inputs, enabling better feature learning.	
UDA + Fine Tune	If you have a related labeled dataset of a different domain but same task (e.g. synthetic), you can perform UDA on that, then fine-tune on your labeled in-domain data.	Can use several domain adaptation techniques.	
Consistency Regularization Stochastic Transformations (NIPS 16,0.5K), Temporal Ensembling (ICLR 17, 12K), Mean Teacher (NIPS17, 17K), VAT (TPAMI 19, 1.3K),	Enforces some priors or assumptions that don't require supervision: • **Label Smoothness**: Datapoints close together in a high-density region should have similar predicted labels • **Clustered Low-Density Separation:** Decision boundary should lie in low density regions, regardless of labels	• **Stochastic Transformation Regularization**(i.e. **Π-Model**)**:** Ensure that 2 data augmentations have similar responses, with supervision when available. For efficiency, you can have a moving average of training sample responses, called **Temporal Ensembling**. • **Mean Teacher** keeps a moving average of previous model parameters, preventing large parameter changes. Supervised when available. • **Virtual Adversarial Training (VAT)**: Applies adv. atk. style optimization on training sample to data augment, then trains on them for predicted label consistency, to make manifold smoother and help decision boundary separation Also: Mixup, entropy regularization	 y_i $\mathbf{x}_i$ stochastic augmentation network with dropout $\mathbf{z}_i$ $\tilde{\mathbf{z}}_i$ cross-entropy squared difference $\mu(t)$ weighted sum loss y Noise Student f_θ EMA Teacher $f_{\theta'}$ Noise $\mathbf{x}$ Cross Entropy d_{MSE} $\mathcal{L}_s$ $\mathcal{L}_u$ Loss $\mathcal{L}_s + \mu(t)\mathcal{L}_u$ δ ε Adversarial Image Minimize KL divergence *Top to bottom: Π-Model, Mean Teacher, VAT.*
Pseudo Labeling (I.e. Label Propagation, self-training) MixMatch (NIPS 19, 1K), FixMatch(NIPS 20, 0.6K)	• Iteratively assigns predicted labels for confident unlabeled samples, and trains on them. Pseudolabeled points chosen can be thresholded, or weighed by certainty. • Allows for better low-density separation between originally unlabeled points, by making the decision boundary more "sharp"	• **MixMatch** holistically combines perturbation regularization, mixup, and pseudolabeling by averaging psuedolabels of K augmentations of an unlabeled sample. **Entropy minimization** is applied to encourage confident predictions on unlabeled data. • **FixMatch** augments weakly and strongly, using only weakly augmented high confidence psuedolabels to supervise strongly augmented.	 Unlabeled Classify ... K augmentations ... Classify Average Sharpen Only keep predictions with high confidence Unlabeled example Weakly-augmented Strongly-augmented Model Model Prediction Pseudo-label Prediction H(p, q) *Top to bottom: MixMatch, FixMatch*







Labeled Data

Unlabeled Data

Pretrained Model

Supervised Loss

Unsupervised Loss

e.g. transformation consistency, (pseudo)label mixup to help enforce assumptions

Question goes here

Answer goes here