Machine Learning

Volume 6

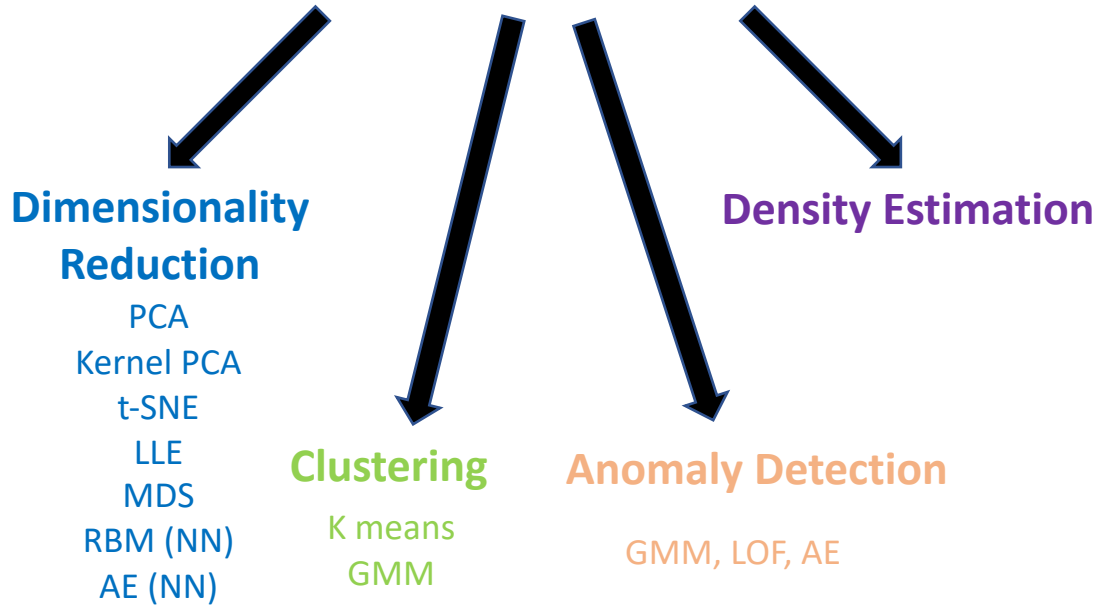
Dimensionality Reduction (DR) Algorithms

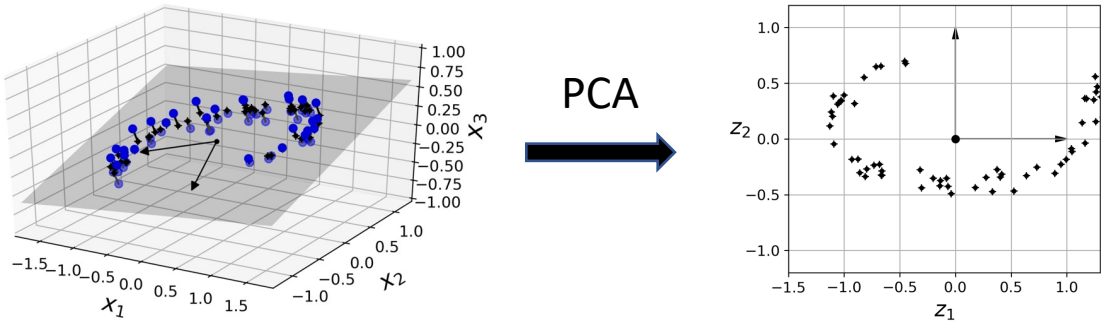
- Why use DR algorithms ?
  - **Problem** : Data distributed sparsely in high dimensions (Curse of dimensionality)
    - ➢ In high dimensional space, training data points are distributed far from each other, and any new test data point will likely be far from the training data making prediction less reliable.
    - ➢ Greater chance of algorithm overfitting because of low volume of training data.
  - DR algorithms combine correlated features into auxiliary variables. This is called feature extraction. Auxiliary variables can then sent to a ML algorithm.
- Two main approaches to dimensionality reduction are **Projection Learning and Manifold Learning**.

Projection Learning

- Example is Principal Component Analysis (PCA).
- Goal is to find a hyperplane onto which the training data can be projected onto. f(z) = Wz
- Cost function minimizes error between the reconstruction term (Wz) and the original data point (x).
- **Minimizing the cost function = Maximizing variance of projected data.**
- Transformation matrix contain the eigen vectors of covariance matrix.

Unsupervised Learning Algorithms

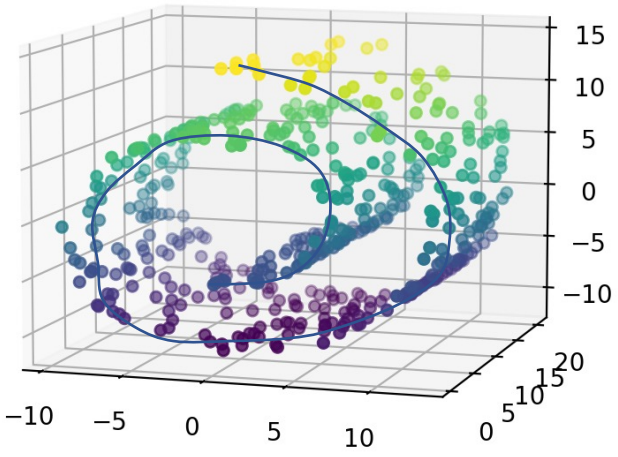


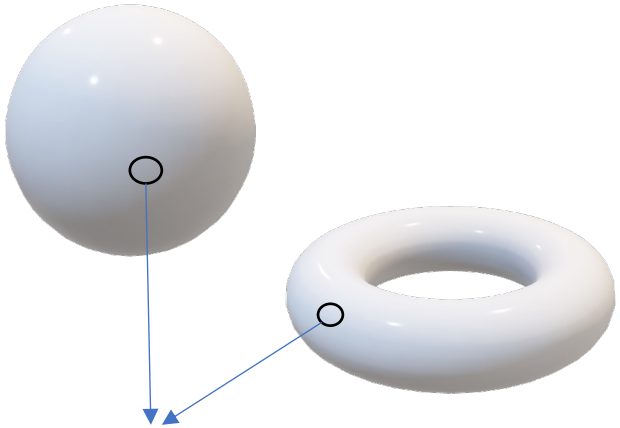


[1] Géron, A. Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow.

Manifold Learning

- A n-manifold is topological space that locally looks like a n-dim. Euclidean space
- d dimension manifold is embedded in n-dimensional space (where d<n).
- Ex : Swiss roll d = 2 and n = 3 (a 2D plane rolled in 3D).
- DR techniques learn the manifold on which training instances lie.
- **Manifold Hypothesis** : Most real world high dim. datasets lie close to a low dim. manifold embedded in the high dim space
- Understanding the topology of the manifold will provide insight on how data is clustered and will also help in generating new data (Generative Learning) (Ex : VAEs)





2D surface locally Euclidean. Sphere and torus are 2D manifolds

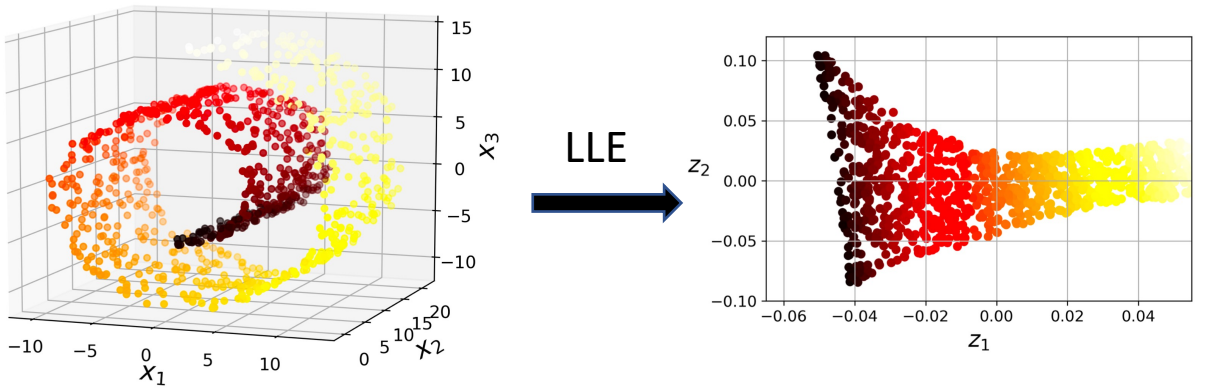
Local Linear Embedding (LLE)

- Introduced in the 2000s. Has similarity to kNN. Uses the locally Euclidean property. (Assumes each data point and its neighbors to lie on a locally linear region of the manifold)
- **Step 1** : Reconstruct each data point from weighted sum of its k nearest neighbors.
- Reconstruction error measured by

$$J = \sum_{k=0}^{n} |X_I - \Sigma W X_J|^2$$

- Weight matrix contains local linear relationships
- **Step 2** : Map training instances onto **d-dimensional space while preserving local relationships**. (d<n).
- Weights learnt from step 1 are kept fixed while choosing appr. low dimensional coordinates that minimize distance between Z and its reconstruction

$$J = \sum_{k=0}^{n} |Z_I - \Sigma W Z_J|^2$$



[1] https://www.kaggle.com/apapiu/manifold-learning-and-autoencoders
[2] Géron, A. Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow.

Manifold mapping

- The manifold hypothesis suggests that many of the high dimensional data used to describe real images lies near a low dimensional manifold. If the data manifold is non trivial (for ex has holes) then an encoder cannot embed it in one-to – one fashion without creating respective low dimensional holes in the latent space as well.
- A differentiable mapping ‘f’ from manifold M to N is said to be diffeomorphism if its inverse mapping N to M is also differentiable. The manifold M and N are called diffeomorphic.
- Lie groups are symmetry groups that are simultaneously differentiable manifolds. They include rotations, translations and scaling. The following paper shows how to construct VAE with latent variables that lie on a lie group.
- Reparameterizing densities on a SO(3) the group of 3D rotations.
- In mathematics a group is a set equipped with a product and the following 4 axioms :
  - Product is closed. (Product of two elements creates a third element that resides within the group)
  - Product is associative
  - There exists an identity element
  - Every element in the set has an inverse.
- Integers equipped with addition operation form a group.
- A lie group (G) has additional structure that is its set is a smooth manifold. Ex: Circle and sphere. We can describe group elements continuously with parameters. The number of parameters defines the dimensions of the group.
- Lie algebra (g) maps of an N-dimensional lie group is the tangent space at the group’s identity which is a vector space of n dimensions. The algebra elements are infinitesimal generators from which all other group elements can be generated.
- Structure of algebra creates a map from element of algebra to vector field on group manifold. This gives rise to the exponential map exp:g -> G which maps algebra element to the group element at unit length from the identity along flow of vector field.

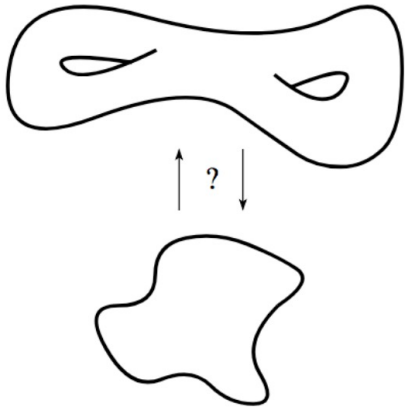


Figure 1.1. An example of problems that arise in mapping manifolds not diffeomorphic to each other. Notice that in the illustrated example the ‘holes’ in the first manifold, prevent a smooth mapping to the second.

[1] Falorsi, L.; de Haan, P.; Davidson, T. R.; De Cao, N.; Weiler, M.; Forré, P.; Cohen, T. S. Explorations in Homeomorphic Variational Auto-Encoding. arXiv July 12, 2018.

Manifold Mismatch and von Mises – Fischer Distribution

- If the useful information in high dimensional data resides close to a hyperspherical manifold then using a gaussian prior will fail to accurately model the data.
- Hyperspherical VAE (S-VAE) is more suitable for capturing data with a hyperspherical latent.
- Imagine a circle dataset $\mathrm{Z} \subset S^1$ where the dataset is embedded in a high dimensional space through a transformation function ($f : Z \subset S^1 \rightarrow X \subset R^N$)
- Traditional AE would quickly recover the latent circle while a normal VAE becomes unstable because of the gaussian prior assumption. This is called as the manifold mismatch.
- **Directional data (such as rotations, translations) are better represented through spherical representations.**
- In low dimensions, gaussian distributions gravitate latent points to be more closer to the origin, encouraging points to cluster in the center. **An ideal prior should stimulate the variance of the posterior without forcing its mean to be close to the center.**
- **In high dimensions, gaussian distributions tend to resemble a uniform distribution on a hyper sphere** with vast majority of mass concentrated on hyper spherical shell.
- The difference here is that in **N-VAE we have a gaussian approximate posterior** and in **S-VAE we have an approximate posterior already defined on a hypersphere.**

Manifold Mapping

- If we let go of the hyperplane assumption then we open up the possibility of using different manifold constructs onto which we can map approximate posterior distributions.
- **A uniform prior would allow data points to spread better over the surface compared to clustering data toward the origin as with having a gaussian prior (N(0,1))**
- In higher dimensions, the cosine similarity is a better metric of measure than compared to Euclidean norm.
- Consider a manifold $\mathcal{M}$ of dimensions $\mathbb{R}^M$ embedded in a high dimensional space $\mathbb{R}^N$. Consider the latent space $\mathcal{F}$ of dimensions $\mathbb{R}^D$.
  - If D <= M then $\mathcal{M}$ and $\mathcal{F}$ are not homeomorphic (a continuous function between topological spaces with a continuous inverse) (i.e there exists no globally continuous and invertible mapping of coordinates from $\mathcal{M}$ to $\mathcal{F}$ and vice versa) and hence enc : $\mathcal{M} \rightarrow \mathcal{F}$ is not a homeomorphism.
  - If D > M then $\mathcal{M}$ and $\mathcal{F}$ are homeomorphic provided sufficiently large D is used. Then enc : $\mathcal{M} \rightarrow emb(M) \subset \mathcal{F}$ is a homeomorphism. However problem that arises is that when sampling random points some of them might not be in emb(M) resulting in poor reconstructions. Also increasing the KL divergence term will cause the emb(M) to spread out more over the hyperplane.

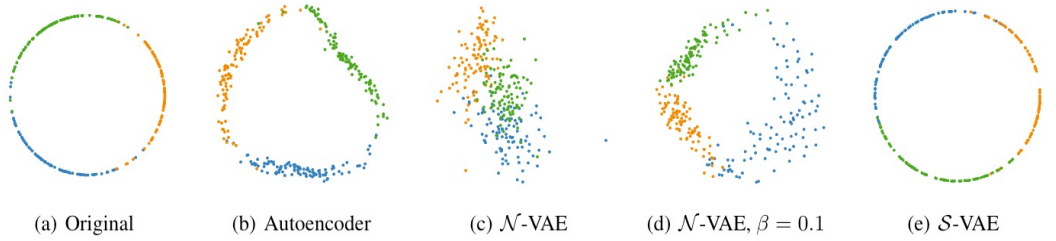


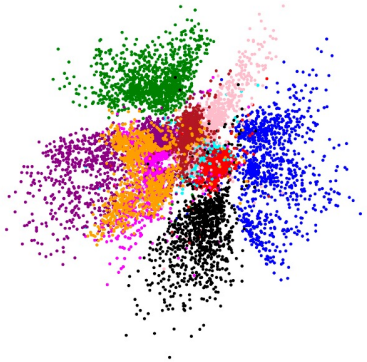
Figure 1: Plots of the original latent space (a) and learned latent space representations in different settings, where β is a re-scaling factor for weighting the KL divergence. (Best viewed in color)

[1] Davidson, T. R.; Falorsi, L.; De Cao, N.; Kipf, T.; Tomczak, J. M. Hyperspherical Variational Auto-Encoders. arXiv September 26, 2018.

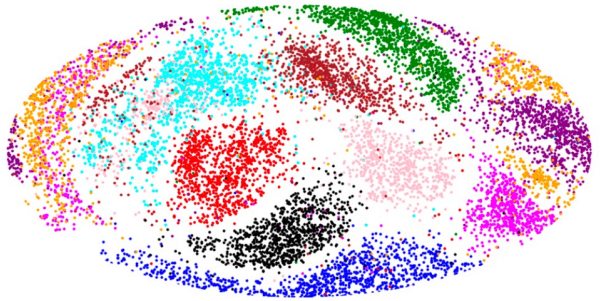
- Von Mises Fischer distribution also called as circular normal distribution or wrapped normal distribution has two important parameters. $\mu \in \mathbb{R}^D$ which indicates the mean direction and κ which is the concentration parameter. When κ = 0 we recover the uniform distribution. The pdf for a random unit vector z $\in$ $\mathbb{R}^D$ or z $\in$ S^{D-1} is defined as
- Now instead of having a gaussian/normal approximate posterior as in N-VAE placed onto a gaussian prior we now have a vMF approximate posterior (circular normal) in S-VAE placed onto a uniform prior U(S^{D-1}) which allows data points to spread better over the wrapped/curved surface. Hence the KL Divergence term now looks like KL(vMF(μ, κ) || U(S^{D-1}))

$$q(\mathbf{z}|\mu,\kappa) = \mathcal{C}_m(\kappa)\exp\left(\kappa\mu^T\mathbf{z}\right),$$

$$\mathcal{C}_m(\kappa) = \frac{\kappa^{m/2-1}}{(2\pi)^{m/2}\mathcal{I}_{m/2-1}(\kappa)},$$



(a) $\mathbb{R}^2$ latent space of the $\mathcal{N}$-VAE.



(b) Hammer projection of $\mathcal{S}^2$ latent space of the $\mathcal{S}$-VAE.

Linear Algebra Prior for understanding SVD

Introduction

- Consider the problem $Ax = b$.
- We can find the values of x by taking inverse of A giving $x = A^{-1}b$.
- We can compute the same by performing Gaussian Elimination which splits A into 3 matrices. $A = LDL^T$ (Decomposition by Elimination)
  - L: upper triangular matrix
  - D: diagonal matrix
- If instead of $Ax = b$ we are interested in solving the problem $Ax = \lambda x$ then we can decompose A into $A = Q\Lambda Q^T$ where Q contains eigenvectors and Λ the eigen values.
  - → Eigen value decomposition
- If A is not square but rectangular, the list of decompositions expand beyond the two mentioned above.
- Solution to a linear system is also the point at which energy is minimized.

minimize $\frac{1}{2}x^TAx - x^Tb$ or solve $Ax = b$.

at minimum $\frac{1}{2}x^TAx = x^Tb$

$\frac{1}{2}Ax = b$

- Gaussian Elimination:

Ex: $2x_1 + 4x_2 = 2$
$4x_1 + 11x_2 = 1$

$$\begin{bmatrix} 2 & 4 \\ 4 & 11 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

$$\Downarrow$$

$$\begin{bmatrix} 2 & 4 \\ 0 & 3 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2 \\ -3 \end{bmatrix}$$

$x_2 = -1$ and $x_1 = 3$ → obtained using back substitution

this is the upper Δle matrix U

lower Δle matrix 'L'

Forward elimination gives the matrix $\begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix}\begin{bmatrix} 2 \\ -3 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$

- $Ux = c$ and $Lc = b$ combining the two we get $LUx = b$ where $A = LU$ (This is the LU factorization)
- For symmetric matrices ($A = A^T$) then $A = LDL^T$
- A symmetric matrix with positive pivots is called a positive definite matrix.
- If some of those pivots are 0 then A is positive semi definite.

Symmetric Matrices

- Consider an order 2 symmetric matrix $A = \begin{bmatrix} a & b \\ b & c \end{bmatrix}$
- We construct the upper Δle matrix U as follows

$$U = \begin{bmatrix} a & b \\ 0 & c - \frac{b^2}{a} \end{bmatrix}$$

Row 2 : Row 2 - (Row 1) $\times \frac{b}{a}$ ← multiplier that will appear in L

- Now we need the lower Δle matrix L. L contains a unit diagonal

$$L = \begin{bmatrix} 1 & 0 \\ b/a & 1 \end{bmatrix}$$

$$A = \begin{bmatrix} 1 & 0 \\ b/a & 1 \end{bmatrix}\begin{bmatrix} a & b \\ 0 & c - \frac{b^2}{a} \end{bmatrix} = \begin{bmatrix} a & b \\ b & c \end{bmatrix}$$

- Divide each row of U by its pivot will factor out a diagonal matrix D

$$U = \begin{bmatrix} a & b \\ 0 & c - b^2/a \end{bmatrix} = \begin{bmatrix} a & 0 \\ 0 & c - b^2/a \end{bmatrix}\begin{bmatrix} 1 & b/a \\ 0 & 1 \end{bmatrix} = DL^T$$

$$A = LU = \begin{bmatrix} 1 & 0 \\ b/a & 1 \end{bmatrix}\begin{bmatrix} a & 0 \\ 0 & c - b^2/a \end{bmatrix}\begin{bmatrix} 1 & b/a \\ 0 & 1 \end{bmatrix} = LDL^T$$

- If A has a symmetric factorization (LDL^T) then it must be symmetric

$$A = LDL^T \qquad A^T = (L^T)^T D^T L^T = LDL^T$$

- $A = LDL^T$ exists for positive definite matrices of all orders.
  - Symmetric matrix with positive pivots
- Instead of looking at the problem $Ax = b$ if we look at $Ax = \lambda x$ then we arrive at another type of factorization for A. Consider instead of 'x' S which contains all linearly indep

$$AS = \Lambda S$$

$$AS = S\Lambda$$

$$A = S\Lambda S^{-1}$$

For orthogonal matrices $SS^T = I$

$S^{-1}SS^T = S^{-1}$ ∴ $A = S\Lambda S^{-1} = S\Lambda S^T$

$S^T = S^{-1}$ $A^T = S^T\Lambda^T(S^{-1})^T = S^T\Lambda S$

Hence if A is symmetric then $A = A^T$ implies that eigen vectors are not only linearly independent but orthogonal as well.

Variance, Covariance and Correlation

- Variance measures spread of data points about mean for a given feature.
- **Co variance** is a measure of directional relationship between 2 variables.
  - The direction of covariance is given by the sign. If covariance is positive then we say the two random variables have positive covariance. If The sign is negative we say the two random variables have negative covariance.
  - If the data is scaled then the covariance changes. This problem is fixed by dividing the covariance with the variance of each feature.
- **Correlation** measures strength and directional relationship between 2 variables
  - Pearson correlation measures the linear relationship between two features.
  - If your data is not linearly related then you should use Spearman Rank Correlation coefficient which establishes a monotonic relationship between two features.

$$\text{Var} = \frac{\sum(x - \bar{x})^2}{n - 1}$$

$$\text{Covar} = \frac{\sum(x - \bar{x})(y - \bar{y})}{n - 1}$$

$$\text{Pearson correlation} = \frac{\text{Cov}(x,y)}{\sigma_x \sigma_y}$$

$$\text{Spearman Rank Correlation coefficient} = 1 - \frac{6 \sum d_i^2}{n(n^2 - 1)}$$

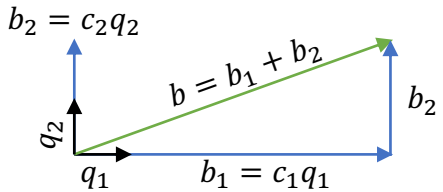
d is the difference in the rank

Orthogonal, Orthonormal and independence

- Consider a vector 'b' in two dimensions. The length of the vector can be computed by taking its square as follows :

$$||b||^2 = bb^T = b_1^2 + b_2^2$$

- Where b_1 and b_2 are the components of b. This is essentially the Pythagoras theorem but the beauty is that it is generalizable to any dimension 'n'.
- Sum of squares of lengths of 'n' components of 'b' and the taking the square root of that sum gives you the length of 'b'. These 'n' components are called orthogonal components.
- If the orthogonal vectors are of unit length then we get a set of orthonormal vectors.
- Any vector can be written as a sum of orthogonal vectors. For example consider our 2D case where we have obtained a set of orthonormal vectors $\{q_1, q_2\}$. We can write our original vector b as follows :



$||q_1|| = 1 \; ||q_2|| = 1$

$$b^2 = bb^T = (b_1 + b_2)^2 = b_1^2 + 2\, b_1 * b_2 + b_2^2 = b_1^2 + b_2^2$$

$b_1 * b_2 = 0$ since they are orthogonal

Ref : Introduction to Linear Algebra by Gilbert Strang

- Note : no matter what constants you use b_1 cannot be written as a linear relation with b_2. If b_1 = cb_2 then the vectors are **linearly dependent and are not orthogonal**
- **All Orthogonal vectors are linearly independent but all linearly independent vectors need not be orthogonal. Example : [0 1] and [1 1] are linearly independent but not orthogonal**
- The two vectors can be combined into a matrix as follows Q = $[q_1 \ \ q_2]$ and c = $[c_1 \ c_2]^T$ then $Qc = b$

$$c = \ Q^{-1}b$$

Since $QQ^T = I$ then $Q^T = \ Q^{-1}$

$$c = Q^T b$$

- Now suppose 'A' is a non orthogonal matrix :

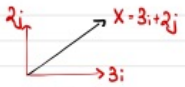
$$\mathrm{A}x = b$$
$$A^T A\ x = A^T b$$
$$x = (A^T A)^{-1} A^T b$$

- Calculating the transpose is much faster that is why we do not multiply by A inverse on both sides but we are still left with computing $(A^T A)^{-1}$
- To orthogonalize A we can use some of the following methods :
- **Gram Schmidt method** or modified Gram Schmidt which gives much more stable results.
  - Gram Schmidt begins with selecting one vector and normalizing to have unit length and uses this as a basis. It then takes the second vector in the set and projects it onto the first to find its component along the direction of the first vector. It then subtracts this component from the second vector to find the vector that is orthogonal to the first. The resulting vector is normalized and now you have two orthonormal vectors. This is repeated for all other vectors in set A and transforming all other vectors such that they are orthogonal to the first one.
  - The above process can be seen as creating two matrices $\mathrm{Q_{nxp}} = [\mathrm{q_1}, \mathrm{q_2}, \mathrm{q_3} \ \ldots \mathrm{q_n}]$which contains all the orthogonal vectors and $\mathrm{R_{nx}}n$ which is a square upper triangular matrix which contains all the constants which will be multiplied by the orthonormal vectors to arrive at vector in A. For example $\mathrm{a_1} = \mathrm{r_{11}q_1}$ and $\mathrm{a_2} = \mathrm{r_{12}q_1} + \mathrm{r_{22}q_2}$ and so on such that A = QR.
- To check if two vectors are truly independent then need to you a combination of statistical and visualization techniques such as mutual information, scatter plot diagrams, Hirschfeld-Gebelein-Rényi maximal correlation which involves calculating the pearsonr correlation after applying different monotonic transformations on the data.

Singular Value Decomposition

Singular Value Decomposition

- Any transformation that preserves the geomtry of the original space is called a linear transformation.
- Consider a vector X in 2D it can be written as a linear combination of 2 basis vectors



OR $X = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$

- Now consider multiplying X with a matrix $A = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$

$$\underset{A}{\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}} \underset{X}{\begin{bmatrix} 3 \\ 2 \end{bmatrix}} = \begin{bmatrix} 0\times3 + 1\times2 \\ 1\times3 + 0\times2 \end{bmatrix} = \underset{X'}{\begin{bmatrix} 2 \\ 3 \end{bmatrix}}$$

- The transformation just interchanges the constants multiplying the basis vectors 'i' and 'j' but still preserves the geometry of the original space. All grid lines wrt to basis vectors are equidistant and llel to each other. Hence the name linear transformation. If the grid lines change due to some function acting on the basis vector we have a nonlinear transformation.
- Any linear transformation can be thought of as stretching, compressing, shearing, reflecting, rotating or any combination of the above to the basis vectors.
- Singular values represent the length and width of the basis

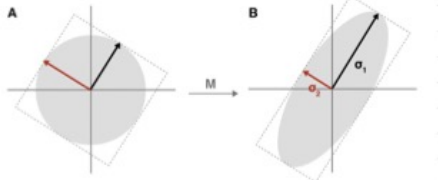


Figure 5: (A) An oriented circle; if it helps, imagine that circle inscribed in our original square. (B) Our circle transformed into an ellipse. The length of the major and minor axes of the ellipse have values σ_1 and σ_2 respectively, called the singular values.

Gram schmidt tells you how to find this orthogonal basis

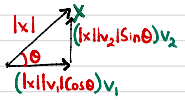
- Any pair of orthogonal vectors form a basis for the space. since any other vector in the space can be written as a linear combination of the 2.
- Consider 2 orthonormal vectors v_1 and v_2. these form the basis set of vectors. A matrix transformation M is applied to this basis set.

$$Mv_1 = \sigma_1 u_1$$
$$Mv_2 = \sigma_2 u_2$$

u_1 and u_2 are unit vectors that are orthogonal.

Note: Can think of v_1 and v_2 lying in the row space that are transformed to column space u_1 and u_2

Any vector 'x' can be written as a sum of these two basis vectors. (v_1 and v_2)



Using the Ale rule of vectors

$$x = (x.v_1)v_1 + (x.v_2)v_2$$

$$Mx = (x.v_1)Mv_1 + (x.v_2)Mv_2$$
$$Mx = (x.v_1)\sigma_1 u_1 + (x.v_2)\sigma_2 u_2$$
$$Mx = \sigma_1 u_1 v_1^T x + \sigma_2 u_2 v_2^T x$$
$$(M - \sigma_1 u_1 v_1^T - \sigma_2 u_2 v_2^T)x = 0$$

* Multiplying M with v_1 moves it into another space where it becomes a unit vector u_1 multiplied with some const σ_1

* Same with $Mv_2 = \sigma_2 u_2$

- For non zero x we get

$$M = \sigma_1 u_1 v_1^T + \sigma_2 u_2 v_2^T$$

$$M = \underbrace{[u_1\ u_2]}_{U}\underbrace{\begin{bmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{bmatrix}}_{\Sigma}\underbrace{\begin{bmatrix} v_1^T \\ v_2^T \end{bmatrix}}_{V}$$

- Important notes:

① v_1 and v_2 are orthonormal and u_1 and u_2 are orthonormal

②

$$M[v_1\ v_2] = \Sigma[u_1\ u_2] \qquad \sigma_1 > 0\ \&\ \sigma_2 > 0$$

$[v_1\ v_2]$ exist in row space $\mathbb{R}^2$; $[u_1\ u_2]$ exist in col space $\mathbb{R}^2$

More general

$$M[v_1\ v_2\ \dots\ v_n] = \Sigma[u_1\ u_2\ \dots\ u_{n-r}]$$

this will have 0's on the diagonal

③ How do we get the orthonormal basis?

- Most matrices are not symmetric, hence their eigen vectors are not orthogonal but are linearly independent
- Start by rewriting the above matrix as

$$M = U\Sigma V^{-1} = U\Sigma V^T$$

- Instead of computing both U and V at once. we only want to compute one of them first. We can do this by calculating MM^T which is symmetric and is positive or positive semi definite.

$$M^TM = V\Sigma^T U^T U\Sigma V^T = V\Sigma^2 V^T$$

- Since M^TM is symmetric and V consists of orthonormal basis, they are essentially the eigenvectors and σ^2 are the eigen values.

- If I want to find the U's then compute MM^T

$$MM^T = U\Sigma V^T V\Sigma^T U^T = U\Sigma^2 U^T$$

(M) ① square but not symmetric ② rectangular

- The main idea is if a matrix is then we can do a factorization of the matrix by calculating MM^T to calculate U and M^TM to calculate V then $M = U\Sigma V^T$. U and V contain Eigenvectors and ortho
- If matrix is square and symmetric from start then $M = Q\Lambda Q^T$ where Q contains the eigen vectors and are orthogonal.
- Eigen values of AA^T and A^TA are the same
- Example:

$$M = \begin{bmatrix} 4 & 3 \\ 8 & 6 \end{bmatrix}$$

row space: $\begin{bmatrix} 4 \\ 3 \end{bmatrix}$; $n(A)$: null space of A

column space: $\begin{bmatrix} 4 \\ 8 \end{bmatrix}$; $n(A^T)$

Orthonormal basis in row space $\left\{ v_1 = \begin{bmatrix} 4/5 \\ 3/5 \end{bmatrix} \quad v_2 = \begin{bmatrix} 3/5 \\ -4/5 \end{bmatrix} \right\}$

Orthonormal basis in col space $\left\{ u_1 = \begin{bmatrix} 1/\sqrt{5} \\ 2/\sqrt{5} \end{bmatrix} \quad u_2 = \begin{bmatrix} 2/\sqrt{5} \\ -1/\sqrt{5} \end{bmatrix} \right\}$

Eigen vectors of $MM^T = \begin{bmatrix} 80 & 60 \\ 60 & 45 \end{bmatrix}$

$$M = \frac{1}{5\sqrt{5}}\begin{bmatrix} 1 & 2 \\ 2 & -1 \end{bmatrix}\begin{bmatrix} \sqrt{125} & 0 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} 4 & 3 \\ 3 & -4 \end{bmatrix}$$

$$(80-\lambda)(45-\lambda) - 3600 = 0$$
$$3600 - 125\lambda + \lambda^2 - 3600 = 0$$
$$\lambda(\lambda - 125) = 0$$
$$\lambda = 0, 125 = \sigma^2$$
$$\therefore \sigma = \sqrt{125}$$

- Essentially we are choosing the right basis for each subspace of a matrix

$v_1, v_2 \dots v_r$ orthonormal basis of row space
$v_{r+1}, v_{r+2} \dots v_n$ orthonormal basis for $n(A)$
$u_1, u_2 \dots u_r$ orthonormal basis for column space
$u_{r+1}, u_{r+2} \dots u_m$ orthonormal basis for $n(A^T)$

SVD finds ortho normal basis for all 4 subspaces

Principal Component Analysis (PCA)

- We consider nxp matrices which are decomposed into nxq matrices by applying SVD.
- Principal components are a sequence of projections of data, mutually uncorrelated and order in variance.
- Consider a set of data in $\mathbb{R}^p$, principal components provide a sequence of best linear approximations to the data of all ranks q< p. This is stored in V_q
- Consider observations (x1,x2, … xn) that belong in $\mathbb{R}^p$. The rank q linear approximation that represents this data is given below:

$$f(\lambda) = \mu + V_q\lambda$$

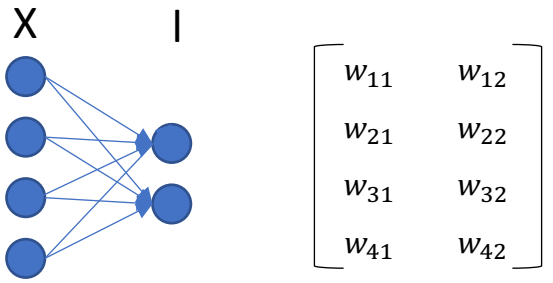
- μ is the location vector ; V_q is a matrix of dimensions (pxq) with q orthogonal vectors ; λ is vector of dimensions 'q' (I.e it is the vector in the rank q space)
- The equation is a parametric representation of an affine hyperplane of rank q. We fit the above affine hyperplane to our data by least squares method.

$$\frac{min}{\mu, \{\lambda_i\}, V_q} \sum_{i=1}^{N} \left|\left| x_i - \mu - V_q\lambda_i \right|\right|^2$$

- If we center our data we can approximate $\mu \approx \bar{x}$. $x_i = \bar{x} + V_q\lambda_i$ then $\lambda_i = V_q^T(x_i - \bar{x})$. This transforms our optimization problem to just optimizing for the 'q' orthogonal vectors.

$$\frac{min}{V_q} \sum_{i=1}^{N} \left|\left| (x_i - \bar{x}) - V_q V_q^T (x_i - \bar{x}) \right|\right|^2$$

- $V_q V_q^T$ is called as the projection matrix and maps each point xi onto the subspace spanned by the columns of V_q.
- Solution to the above optimization problem gives you SVD.



PCA computes an orthogonal set of eigen basis vectors however a vanilla autoencoder cannot because there is no information inbuilt in model or loss function to maximize the variance of the projected data

Non linear Principal Component Analysis (PCA)/ kernel PCA

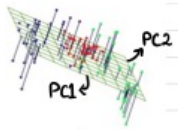
Basics of Principal Component Analysis

- Autoencoders are a class of feed forward neural networks
- They comprise of 3 parts: Encoder, Code & Decoder
  Auto Encoders try to regenerate the input from reduced dimension / latent space of variables. They are unsupervised learning technique. We are not trying to map function from feature space (x) to target space (y). Goal is to minimize the reconstruction loss

Input → Encoder → Code → Decoder → Output

- PCA learns linear transformation of data, while Auto Encoders are capable of learning non linear manifolds (↳ learn linear manifold)

Projecting from 2D to 1D



Projecting from 3D to 2D

The linear separating hyper plane is represented by $f(z) = \mu + V_q \lambda$ ← Parametric rep of affine hyperplane
μ → location vector
V_q → $p \times q$ matrix with q orthogonal unit vectors
λ → q vector of parameters

Data lies in $\mathbb{R}^D$ space which is reduced to lie in $\mathbb{R}^L$ space.
$\omega_j \in \mathbb{R}^D$, $x_i \in \mathbb{R}^D$ and $z_i \in \mathbb{R}^L$

- Note: The derivation assumes that 'x' data is standardized. (i.e data is centered, mean subtracted from each feature x_i ('d' dimensional vector) and normalized)

These are the eigen vectors of the cov. matrix
Principal components are uncorrelated and ordered in terms of variance given by their eigenvalues

- If normalization is not performed then PCA will try to project all the data onto the axis for the feature which has large range in its numerical values. This ignores the contribution of other features in maximizing variance.
- For the purpose of derivation let us consider projecting 'd' dimensional data onto 1 dimension where the axis is ω_1. Reconstruction error $J(\omega_1, z_1)$ is as follows:

$\omega_1 = \begin{bmatrix} l_1 \\ l_2 \\ l_p \end{bmatrix}_{p \times 1}$ $l \in \mathbb{R}^p$ $\quad \omega_1^T = [l_1, l_2 \dots l_p]_{1 \times p}$

$\omega_1^T \omega_1 = l_1^2 + l_2^2 + l_3^2 \dots l_p^2 = 1$
since ω_1 is a unit vector

$$J(\omega_1, z_1) = \frac{1}{N} \sum_{i=1}^{N} \| x_i - \omega_1 z_{1i} \|^2 = \frac{1}{N} \sum_{i=1}^{N} (x_i - \omega_1 z_{1i})^T (x_i - \omega_1 z_{1i})$$

$$= \frac{1}{N} \sum_{i=1}^{N} \left(x_i^T x_i - x_i^T \omega_1 z_{1i} - \omega_1^T z_{1i}^T x_i + \omega_1^T z_{1i}^T \omega_1 z_{1i} \right)$$

$\omega_1^T \omega_1 = 1$
$-2\omega_1^T x_i z_{1i}$
since in 1D space this is just z_{1i}^2

This is reconstruction of input from latent space

$$J(\omega_1, z_1) = \frac{1}{N} \sum_{i=1}^{N} \left(x_i^T x_i - 2 x_i \omega_1^T z_{1i} + z_{1i}^2 \right)$$

- Now lets take derivative of $J(\omega_1, z_1)$ wrt z_{1i} ...

$$\frac{\partial J(\omega_1, z_1)}{\partial z_{1i}} = \frac{1}{N} \left[-2 x_i \omega_1^T + 2 z_{1i} \right] = 0$$

↳ specific instance of training data

$z_{1i} = x_i \omega_1^T$ ($1 \times p$ · $p \times 1$ → 1×1)

- Now we know what z_{1i} should be in order to minimize the reconstruction error. It is obtained by orthogonally projecting x_i onto ω_1 (the first principal component). Substituting z_{1i} back into cost function gives

$$J(\omega_1) = \frac{1}{N} \sum_{i=1}^{N} \left(x_i^T x_i - 2 z_{1i}^2 + z_{1i}^2 \right) = \frac{1}{N} \sum_{i=1}^{N} \left(x_i^T x_i - z_{1i}^2 \right)$$

$$J(\omega_1) = \underbrace{\frac{1}{N} \sum_{i=1}^{N} x_i^T x_i}_{X^T X = \text{cor matrix} \cdot \text{constant}} - \underbrace{\frac{1}{N} \sum_{i=1}^{N} z_{1i}^2}_{\text{variance of projected data}}$$

(mean of projected data = 0 since x_i are centered)

- Hence minimizing $J(\omega_1)$ is equivalent to maximizing variance of projected data

$$\text{Var}[z_1] = \frac{1}{N} \sum_{i=1}^{N} z_{1i}^2 = \frac{1}{N} \sum_{i=1}^{N} z_{1i} z_{1i}^T = \frac{1}{N} \sum_{i=1}^{N} x_i \omega_1^T x_i^T \omega_1 = \omega_1^T \hat{\Sigma} \omega_1$$

(for higher dim) — generalization

$\hat{\Sigma}$ = correlation matrix ↳ positive semi definite

- Now it is possible to maximize variance by letting $\omega_1 \to \infty$ but we know ω_1 is a unit vector. Hence we maximize $\text{Var}[z_1]$ subject to constraint that $\|\omega_1\| = 1$. We can formulate this using Lagrangian multipliers.

$$L = f(x, y \dots) - \lambda_1 [g(x, y \dots) - c]$$

$$L = \text{Var}[z_1] - \lambda_1 [\omega_1^T \omega_1 - 1] = \omega_1^T \hat{\Sigma} \omega_1 - \lambda_1 [\omega_1^T \omega - 1]$$

$$\frac{\partial L}{\partial \omega_1} = 0 = 2\hat{\Sigma}\omega_1 - 2\lambda\omega_1 = 0 \quad (\text{OR}) \quad \hat{\Sigma}\omega_1 = \lambda_1 \omega_1$$

Hence the vector along which the variance of projected data is maximum is actually the eigen vector of the correlation matrix $\hat{\Sigma}$. λ_1 is the eigen value.

Lagrange Multipliers

- Given a function $f(x, y \dots)$ which we want to maximize or minimize subject to constraint that another multivariate function is constant ($g(x, y \dots) = c$), do the following steps
- Introduce new variable 'λ' called Lagrangian multiplier and define new function L as follows:

$$L = f(x, y \dots) - \lambda [g(x, y \dots) - c] \quad (L \text{ is called Lagrangian})$$

- In this derivation we find the orthogonal basis vectors which define a hyperplane onto which we project our data. This is given by matrix V_q.
- We find this basis by first constructing a symmetric matrix. This is done by calculating XX^T
- Now performing SVD is essentially an Eigen Value decomposition. The eigen values would be squares of the singular values, the resultant eigen vectors will be orthogonal. **If matrix not square and symmetric then we cannot obtain orthogonal eigen vectors.**
- **Its should be noted that the SVD is linear transformation technique that produces orthonormal basis vectors hence they are linearly independent but could be dependent through some non linear relationship.**
- Pearson correlation analysis of points in the latent space will yield identity matrix.
- For stronger tests for independence look at techniques mentioned in previous slides.
- Kernel PCA transforms the points to a higher dimensional space where linear projection techniques like PCA can be performed.

Linear Discriminant Analysis

- Finds a linear hyperplane that separates 2 or more classes.
- Resulting linear combination can be considered as dimensionality reduction before proceeding for classification.
- Optimization objective is to maximize the variance between classes while minimizing variance within a class.
- Closely related to Analysis of Variance (ANOVA) which

t-Stochastic Embedding (t-SNE)

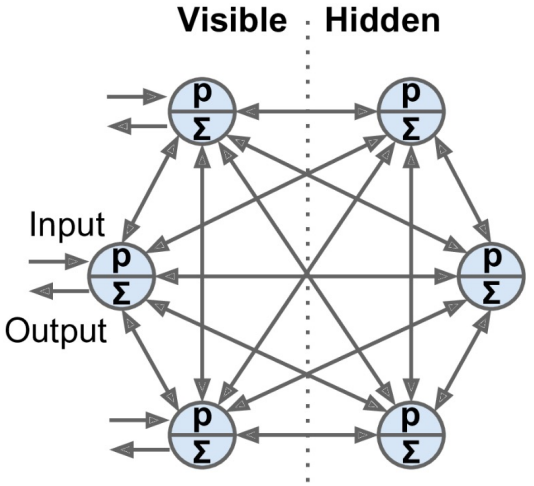
Uniform Manifold Approximation and Projection (UMAP)

Independent Component Analysis (ICA)

Neural Net architecture based Unsupervised Learning

Boltzmann Machine (Directed Graph)

- Fully connected ANNs but based on stochastic units. Each outputs 0 or 1 with a probability based on the sigmoid function.
- Using the training data, BMs learn appropriate weights to generate output vectors that resemble the training vectors.
- Visible units receive input and are from where outputs are read. Because of stochastic nature BM will never settle into a fixed configuration.
- After many iterations, the probability of being in a particular configuration will only be a function of the connection weights and bias and not of the original configuration. When the network reaches this state it is said to be in thermal equilibrium.
- No method available to train a Boltzmann Machine.



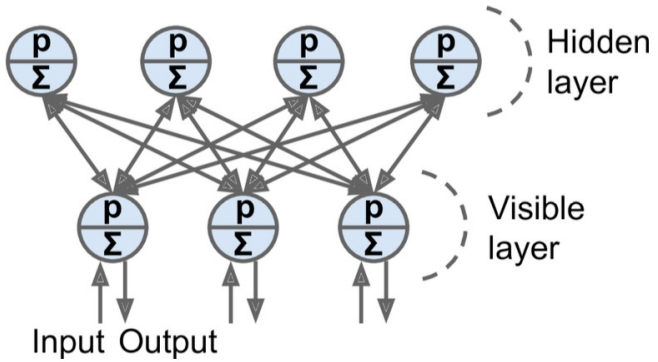
Stochastic equation

$$z_i = b_i + \sum_{j=1}^{N} w_{i,j}\, s_j \qquad p(s_i = 1) = \sigma\left(\frac{z_i}{T}\right)$$

S_j is the ith neuron state (0 or 1)
wij is connection weight between i and j ,
bi is ith neuron bias
N is number of neurons in network
T is temperature of network

Restricted Boltzmann Machine (Undirected Graphs)

- A Boltzmann machine where there are no connections between the visible and hidden units. (Latent variables are conditionally independent of input variables)
- **Unlike BM, RBM's can be trained using Contrastive Divergence.**
- Compute the state of hidden variables by passing visible units through stochastic equation. This generates h. Using h generate x' (reconstruction). Then generate h' using x'. Process repeats
- RBM's learn the probability distribution of the inputs.



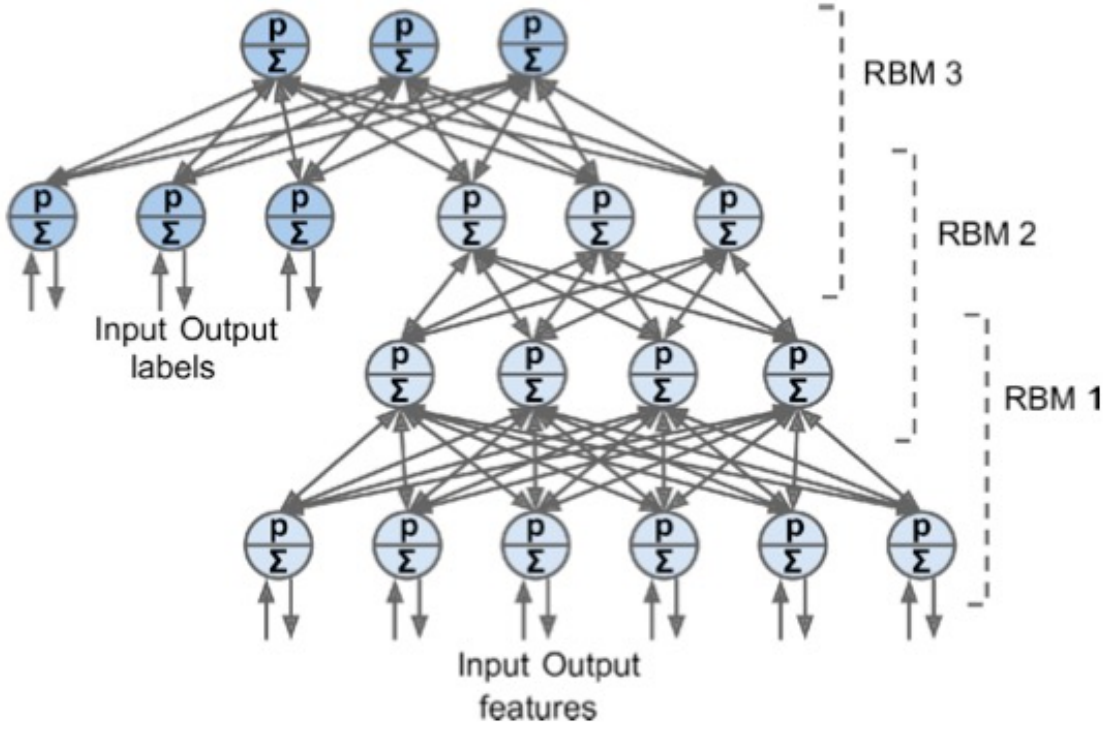
BM and RBM use energy functions to determine if the network is learning

[1] Hinton G. (2014) Boltzmann Machines. In: Sammut C., Webb G. (eds) Encyclopedia of Machine Learning and Data Mining. Springer, Boston, MA. https://doi.org/10.1007/978-1-4899-7502-7_31-1
[2] Géron, A. Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow

Neural net architecture based Semi Supervised Learning

Deep Belief Nets

- Obtained by stacking several layers of RBMs.
- Was useful in the early days of Deep Learning when training deep networks was difficult and labelled data was less
- The model is trained layer by layer (greedy learning) using contrastive divergence like in RBMs.
- DBNs reproduce the probability distribution of the inputs just like RBMs but are more powerful as they take advantage of the fact that real world data is hierarchical (lower layers learn low level features while higher layers learn more complex features).
- Visible units can be added to the higher layers to associate high level features learned from the training data with training labels.
- Inputs can flow both ways through a DBN.
    - Forward direction : Given a test instance information flows through network and generates an output label (classification).
    - Reverse direction : Activating a label, the information flows backward all the way up the network to generate an input.



[1] Hinton, G. Deep Belief Nets. In *Encyclopedia of Machine Learning and Data Mining*; Sammut, C., Webb, G. I., Eds.; Springer US: Boston, MA, 2014; pp 1–4. https://doi.org/10.1007/978-1-4899-7502-7_67-1.
[2] Géron, A. Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow.

- Old ML algorithms such as RBMs use energy functions to determine if model is performing well or bad. Kind of like how newer models use loss functions to determine if network is learning.
- A simple example is the Binary RBM for which the energy function is given below. v and h represent the states of the visible and hidden. b_r, c_k are the bias terms of the visible and hidden units and W_{rk} is the weight matrix. All the params can be combined into θ.

$$E(s|\theta) = -\left(\sum_{r=1}^{R} v_r b_r + \sum_{k=1}^{K} h_k c_k + \sum_{r=1}^{R}\sum_{k=1}^{K} v_r h_k W_{rk}\right)$$

- A vector containing the states of all the units in the BM is called state vector (s)
- m contains all possible binary state vectors

$$p(s|\theta) = \frac{e^{-E(s|\theta)}}{\sum e^{-E(m|\theta)}} = \frac{e^{-E(s|\theta)}}{Z}$$

Difficulty is in calculating the second term which contains the normalization constant Z(W) which is sum of exponential terms

$$\frac{dlogP(s|\theta)}{dW} = -\frac{dE(s|\theta)}{dW} - \frac{dlogZ}{dW} \qquad \frac{dE(s|\theta)}{dW} = -v_r h_k$$

$\langle v_r h_k \rangle_{data}$ is the expected value in the data distribution
$\langle v_r h_k \rangle_{model}$ is the expected value when BM samples state vectors from its equilibrium distribution

$$\left\langle \frac{dlogP(s|\theta)}{dW} \right\rangle = \langle v_r h_k \rangle_{data} - \langle v_r h_k \rangle_{model}$$

RBM Theory

IN RBMs since no connection between hidden units they are conditionally independent

$$p(\mathbf{d}|\theta_1 \ldots \theta_n) = \frac{\Pi_m p_m(\mathbf{d}|\theta_m)}{\sum_{\mathbf{c}} \Pi_m p_m(\mathbf{c}|\theta_m)}$$

In CD we start the chain at data distribution p_o and run the chain for a small number of steps. $p_n(x;W) = p_n = T^n p_o$ is the nth step. Distribution in the Markov chain with transition matrix T.

Gradient Ascent $$w_{ij}^{(\tau+1)} = w_{ij}^{(\tau)} + \eta\left(\left\langle \langle y_i x_j \rangle_{p(\mathbf{y}|\mathbf{x};\mathbf{W})} \right\rangle_0 - \langle y_i x_j \rangle_n\right). \quad W_{ij}(n+1) = W_{ij}(n) + \alpha \frac{dL(W;X)}{dW_{ij}} = W_{ij}(n) + \alpha \frac{d<logp(X;w)>}{dW_{ij}}$$

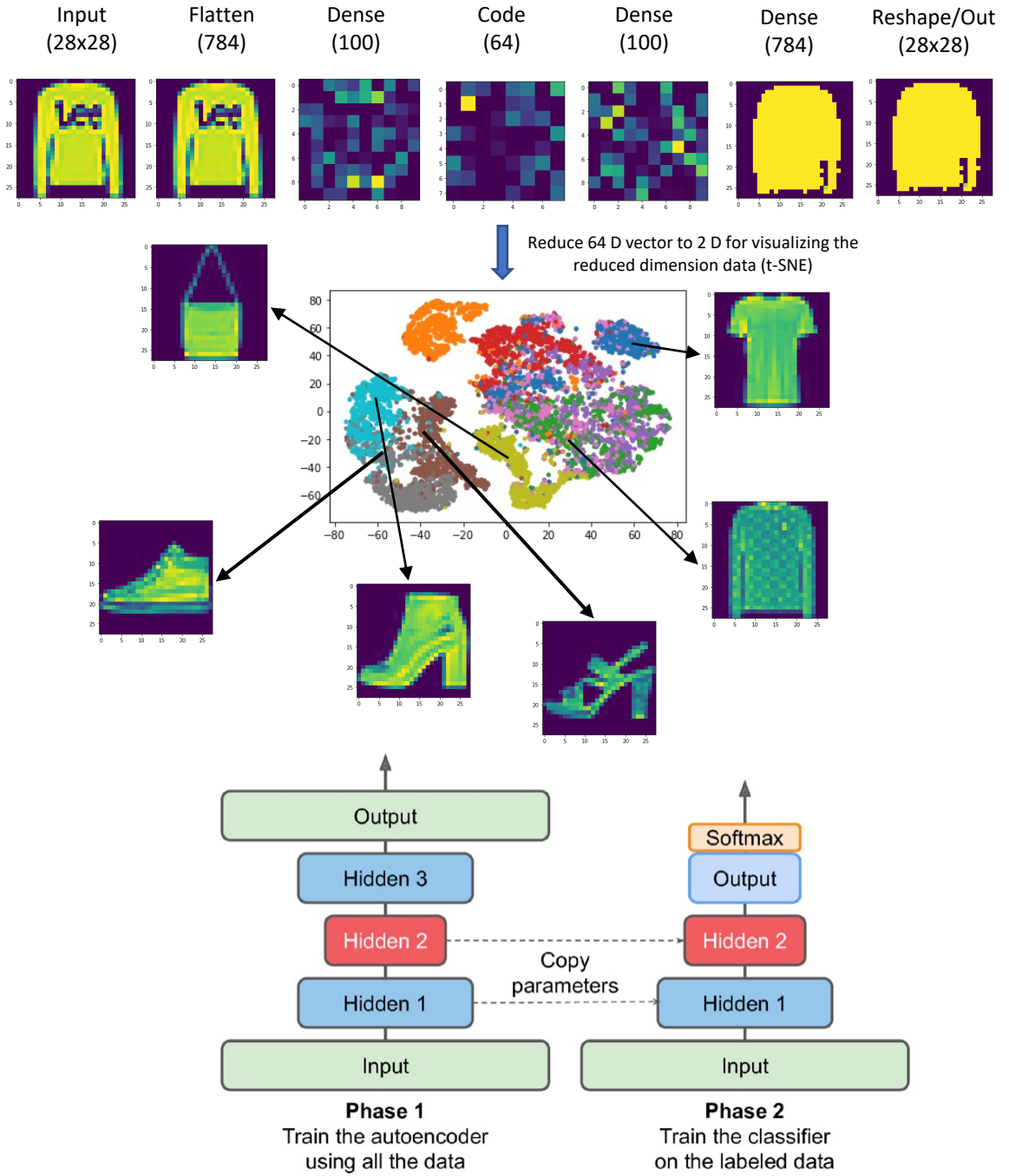
[1] Carreira-Perpinan, M. A.; Hinton, G. E. On Contrastive Divergence Learning. 8.
[2] Salakhutdinov, R.; Mnih, A.; Hinton, G. Restricted Boltzmann Machines for Collaborative Filtering. In *Proceedings of the 24th international conference on Machine learning - ICML '07*; 2007; pp 791–798. https://doi.org/10.1145/1273496.1273596.
[3] Hinton G. (2014) Boltzmann Machines. In: Sammut C., Webb G. (eds) Encyclopedia of Machine Learning and Data Mining. Springer, Boston, MA. https://doi.org/10.1007/978-1-4899-7502-7_31-1

Autoencoders (Dimensionality Reduction + Feature Extraction)

- Autoencoders
  - Learn latent space representations (codings) of the input data (DR).
  - Can generate new data based on learned codings (Generative Learning Ex : GANs)
  - Act as feature detectors for pretraining deep NN
- AEs learn to replicate input from latent space representations (bottleneck in NN – Stacked AE) and/or by introducing noise (missing data/erroneous values) into the inputs (Denoising AE).
- Unlike in RBM and DBN, AE does not use stochastic nodes or energy functions and is trained using one complete forward and backward pass through the network.
- Stacked AE can learn more non linear mappings.
- Weights of decoder can be tied to that of encoder to reduce training time.

Autoencoders for pre training Supervised ML models

- The autoencoder works on the entire dataset (labelled + unlabelled).
- The low level features learned by AE can be transferred to a neural net for training on labelled data.
- Classifier does not need to extract the low level features from the limited training data. It uses the features captured by training the AE.
- Popular applications of Autoencoders
  - Training deep learning networks (The encoder part in an AE acts like a classifier)
  - Compression (Latent space generation)
  - Classification (Using latent space to classify – Semi supervised learning)
  - Anomaly detection (Denoising AE)
  - Generative Models (Organized and continuous latent space generators – VAEs)
- Types of Autoencoders : (Simple, Sparse, Deep, Convolutional (dealing with images), Denoising , Variational



Why use Autoencoders ?

- There are 100s of inputs that go into making perovskite solar cell device.
- Examples of these inputs are choice of chemicals (A,M,X precursors, type of solvents, type of antisolvents, additives), perovskite film features (MHP film processing conditions, conductivity measurement parameters etc.), perovskite solar cell device feature (type of HTM, ETM, contacts etc) and each of these influence measurement of properties across various scales except the atomic scale properties (for ex: solvent coordination) which does not depend on features such as spin coating speed, type of HTM, ETM but a macro scale property such as PCE, device stability can and maybe will have dependency on features from lower scales.
- A 100 D space is a very sparse structure with data points situated very sparsely within the space. For learning algorithms like machine learning to make good predictions a lot of data is needed. Exactly how much ? See page to the side.
- Hence dimensionality reduction is important and various algorithms to do that. (PCA)
- We strive to use a neural network approach using a network architecture called Autoencoders.
- Autoencoders are efficient feature extractors that are great for dimensionality reduction.
- Different autoencoder implementations structure the latent or compressed space differently.
- Vanilla Autoencoders work without supervision and hence their latent structure is filled with gaps and overlap between classes.
- Supervised autoencoders add a supervised label to the latent to help structure the latent more effectively. Multiple supervised losses can be tagged to structure the latent that is conducive for regression or classification tasks for both labels.
- Convolutional autoencoders are more effective in dealing with images or even sequence data where there is a repetitive structure within the image or sequence data that can be captured by the use of convolutional kernels.
- Variational Autoencoders are a whole different type of model where the latent is regularized to behave like a probability distribution.

Curse of Dimensionality

Problem 1 : To cover even a very tiny fraction of the high D space we need to cover a wide range of each input var.

- Consider a unit line, square, cube ... n-D cube
- Now consider e to denote edge length
- r = Volume fraction Ex: $r_1 = \frac{e_1}{1}$; $r_2 = \frac{e_2^2}{1}$; $r_3 = \frac{e_3^3}{1}$ etc $\quad r_p = \frac{e_p^p}{1}$
- $e_p(r) = \frac{(r_p)^{1/p}}{1}$
- For $p=1$ $e_1(0.1) = 0.1$; $e_1(0.63) = 0.63$; $e_1(0.99) = 0.99$
  $p=10$ $e_{10}(0.1) = (0.1)^{0.1} = 0.794$; $e_{10}(0.63) = (0.63)^{0.1} = 0.955$; $e_{10}(0.99) = (0.99)^{0.1} = 0.999$

To capture 10% to 90% of the data in 10D cube we must cover 80% to 99% range of each input variable.

$p=100$ $e_{100}(0.01) = 0.01^{0.01} = 0.955$%.

To capture just 1% of the local neighbourhood we must cover 95% range of each input variable

Problem 2: Data points lie close to boundary of high D space compared to the volume.

- Consider a p dim hyper sphere with radius 'r'. Volume is then r^p
- Consider a 'p' dim hypersphere within the original hypersphere with radius k_r.
- Volume of this hyper sphere is $(k_r)^p$.
- Difference in Volumes = $r^p - (k_r)^p$
- Proportion of volume greater than 'k_r' is $= \frac{r^p - (k_r)^p}{r^p} = 1 - (k)^p$
- Probability that all 'N' randomly sampled points have distance greater than 'k_r' from origin is $[1-(k)^p]^N$
- Setting the probability to 50%

$$\left(1 - \left(\frac{1}{2}\right)^{\frac{1}{N}}\right)^{1/p} = k$$

Taking $p=100$ and $N=500$ then $k = 0.9363$

Semi supervised learning

- Dataset contains majority of unlabeled data with some labelled data.
- Example is DBN which is composed of RBM's which are trained seq. in unsupervised manner and then whole system is fine tuned using supervised learning techniques.

Semi supervised learning with Autoencoders

1. Learning compact document representations

- Input fed to encoder to produce code. Input is reconstructed from generated code. To incorporate labelled information a classifier is included in the feedback module.
- During training, the parameters of the encoder, decoder and classifier are learned by minimizing $E_R + \alpha E_C$ (E_R – reconstruction error ; E_C – classification error ; α – balancing coefficient)
- E_R ensures the model learns the dependencies and structure within the input data while E_C ensures that the lower dimension code can be used for discriminating between classes.
- When input sample not accompanied with a label then the loss collapses to simply E_R .
- **Purely unsupervised model will fail to capture the information in the codings that is relevant for classification/regression.**
- Restricted Boltzmann Machines used for retrieval and clustering. Also they are reported to have robustness to uncorrelated noise in the input as they attempt to model the input distribution. Rely on unsupervised pre traning using Contrastive Divergence, followed by supervised back propagation. Also difficult to predict when to stop training and how long Markov Chain must run.
- Authors chose to go with Autoencoders. Build a deep network with building blocks as autoencoders.
- Learn model weights using both a supervised and unsupervised objective.
- Document classification : previously documents were classified based on a vector of word counts which does not capture dependencies between related words or synonyms. Here the authors aim to find a document representation based on stacked semi supervised autoencoders.
- Authors used different types of representations sparse and a dense encoding .
- Input to first autoencoder is a vector of word counts (sparse input).
- The way the model works is that each layer is an encoder which produces a code that is propagated to the next layer.
- The decoder of the first layer is a Poisson regressor as we aim to reconstruct the vector of word counts.

Ranzato, M. A.; Szummer, M. *Proceedings of the 25th international conference on Machine learning - ICML '08*; 2008 https://doi.org/10.1145/1390156.1390256.

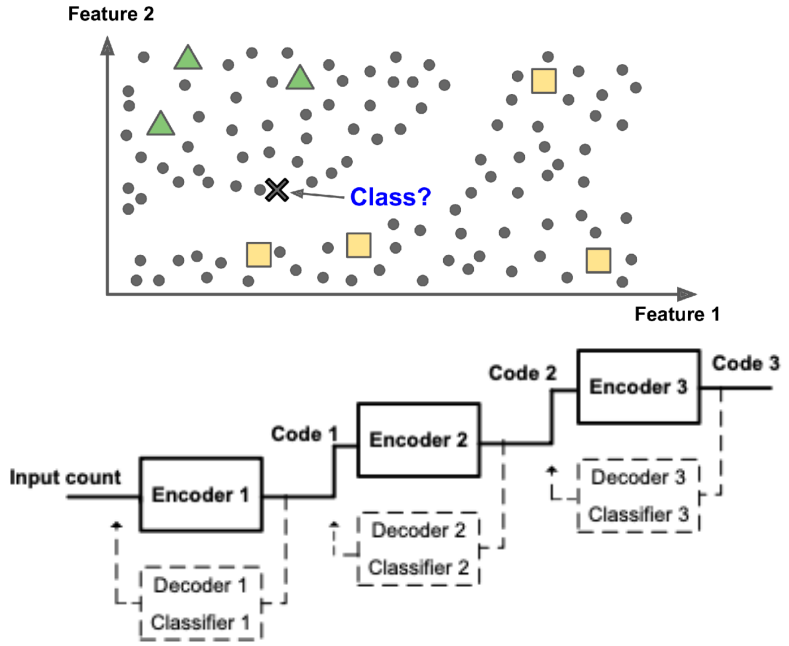


Figure 1. Architecture of a model with three stages. The system is trained layer by layer. During the training of the n-th layer, the n-th encoder is coupled with the n-th decoder and classifier (shown in dashed line). The n-th encoder will provide the codes to train the layer above. After training, the feedback decoding modules are discarded and the system is used to produce very compact codes by a feed-forward pass through the chain of encoders.

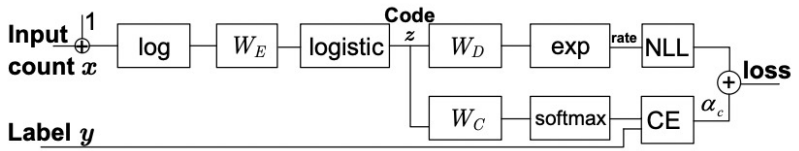


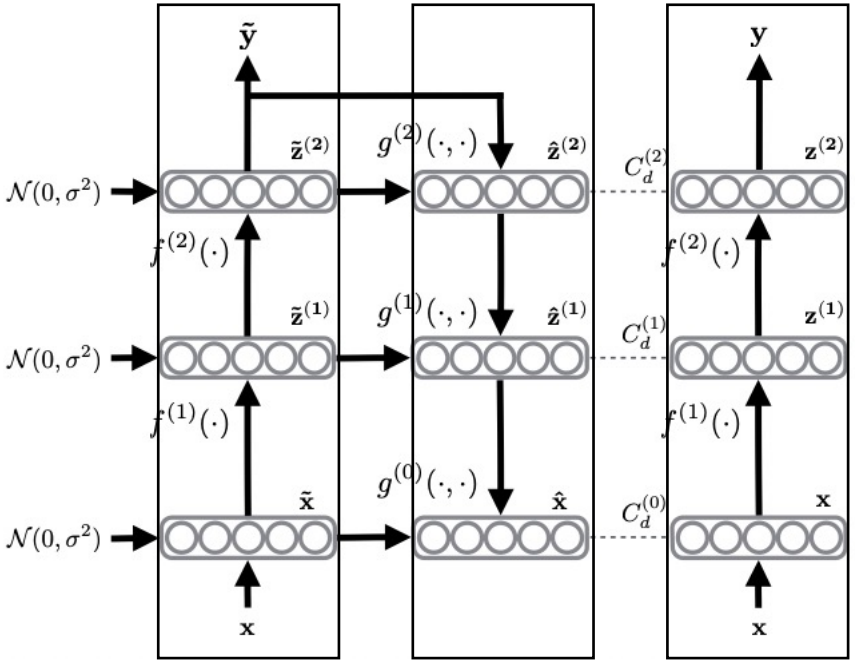
Figure 2. The architecture of the first stage has three components: (1) an encoder, (2) a decoder (Poisson regressor), and (3) a classifier. The loss is the weighted sum of cross-entropy (CE) and negative log-likelihood (NLL) under the Poisson model.

2. Ladder Architecture using Denoising Autoencoders

- Hierarchical latent variable models allow lower layers to focus on the details while the higher layers focus on more abstract representations.
- In denoising AE, the model is trained to reconstruct input (x) from a corrupted version of the input ($\tilde{x}$). Loss is then $\| x - \hat{x} \|^2$ where $\hat{x}$ is the reconstructed version of input.
- The Denoising Source Separation (DSS) method involves using denoising functions $\hat{z} = g(z)$ of the latent variables to train the mapping $z = f(x)$. Loss is given by $\| z - \hat{z} \|^2$.
- Architecture :
- Skip connection between encoder and decoder at each level allows information to be passed between them. This allows the encoder to be trained through the cost function at each intermediate level to recognize important features in the corrupted input that match the original distribution of z better.
- Cost function C minimizes the error between the reconstructed $\hat{z}^l$ and actual z^l (from uncorrupted encoder).

$$C \leftarrow C + \sum_{l=0}^{L} \lambda_l \left\| \mathbf{z}^{(l)} - \hat{\mathbf{z}}_{\mathrm{BN}}^{(l)} \right\|^2$$

- At each level (l) of decoder : $\hat{z}^{l+1} = g(\hat{z}^l, \tilde{z})$ (g is the denoising function).
- Mappings shared between corrupted and uncorrupted encoders. Noise added at each level of corrupted encoder.
- The ladder network depicts nested denoising autoencoders.



Encoder using corrupted input ($\tilde{x}$)

Decoder using

Encoder using Original input (x)

Uncorrupted encoder : x -> z^1 -> z^2 … -> y

Corrupted encoder : x -> $\tilde{z}^1$ -> $\tilde{z}^2$ … -> y

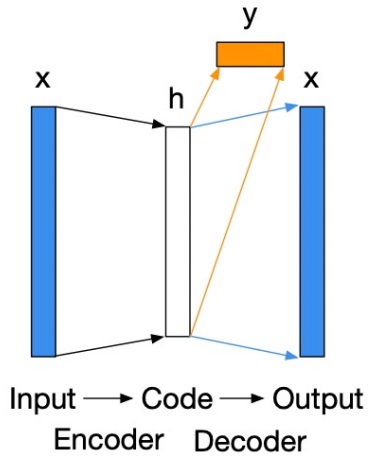
Decoder : $\tilde{z}^l$+ $\hat{z}^{l+1}$ -> $\tilde{z}^{l-1}$… -> $\hat{x}$

[1] Rasmus, A.; Berglund, M.; Honkala, M.; Valpola, H.; Raiko, T. Semi-Supervised Learning with Ladder Networks. 9

Supervised Autoencoders

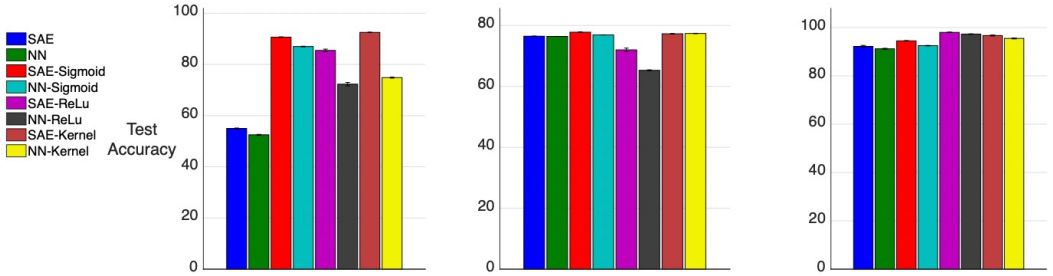
- Supervision occurs in the latent space.
- Network can predict both inputs and outputs.
- Linear supervised Autoencoders are uniformly stable. (Little difference between models learned from any sub sample of data). In other word performs well on unseen data. Good generalization capabilities.
- Le et al. show that including reconstruction error leads to better generalization performance. [1] Treat the reconstruction error as a regularizer to promote stability.

$$L(\mathbf{F}) = \frac{1}{t}\sum_{i=1}^{t} L_p\left(\mathbf{W}_p\mathbf{F}\mathbf{x}_i, \mathbf{y}_{p,i}\right) + L_r\left(\mathbf{W}_r\mathbf{F}\mathbf{x}_i, \mathbf{y}_{r,i}\right).$$

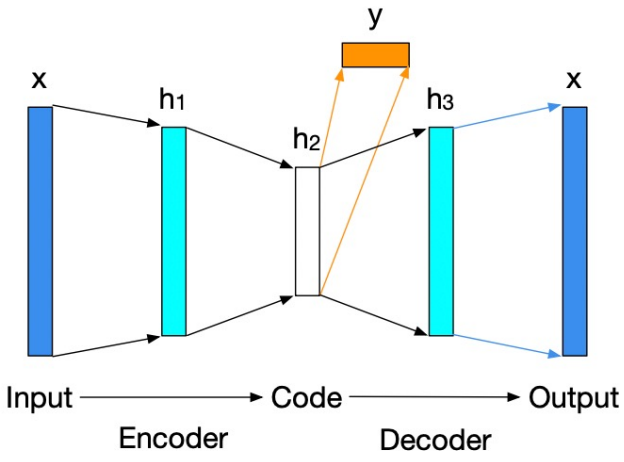


(a) (Linear) Supervised Autoencoder

- Authors divide learning into 2 separate tasks. Primary task is supervised learning with the corresponding loss function Lp and weights Wp and auxiliary task is reconstruction of input with corresponding loss function Lr and with weights Wr.



(a) Deterding dataset (b) SUSY dataset (c) MNIST dataset



(b) Deep Supervised Autoencoder

- In the simple unsupervised learning case there is nothing to guide the process of creating a proper latent space. Adding in a supervision task helps create a more organized and informed latent space.[1]
- Supervised autoencoders offer benefit of **Multi Task Learning (MTL)** as well as **Dimensionality Reduction (DR)** to extract important features from inputs.

[1] Le, L.; Patterson, A.; White, M. Supervised Autoencoders: Improving Generalization Performance with Unsupervised Regularizers. 11. (2018)

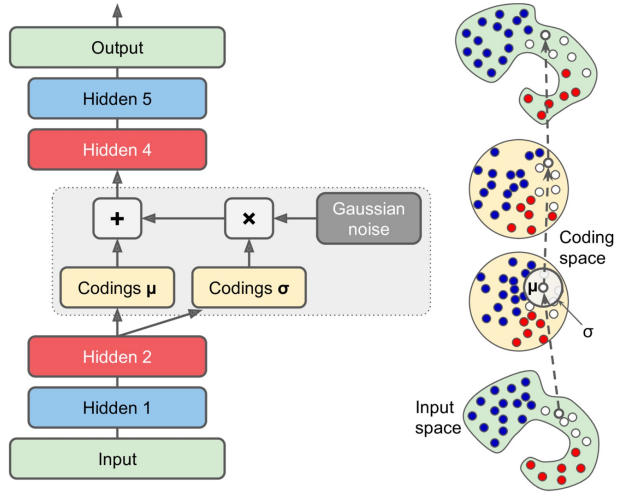
[1] Mohri, M.; Rostamizadeh, A.; Storcheus, D. Generalization Bounds for Supervised Dimensionality Reduction.

Variational Autoencoders

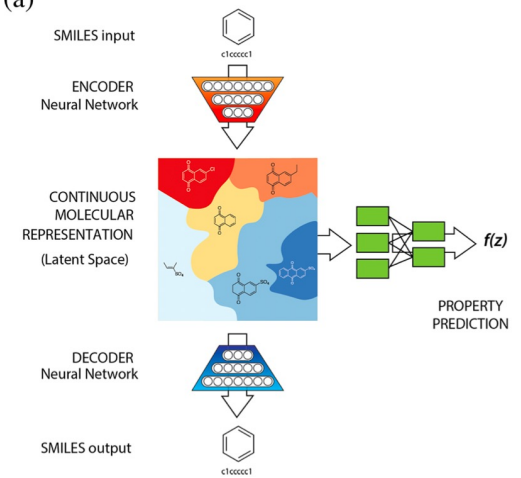
- Introduced in 2013 by Diederik Kingma and Max Welling.
- They are probabilistic autoencoders. (Output determined by chance even after training).
- They are generative autoencoders. (They can generate new instances that looked like they were sampled from the training set)
- Instead of producing coding directly for a given input, the encoder produces a mean coding (μ) and standard deviation (σ). The decoder decodes the sampled coding normally.
- Once training is done, a new instance can be generated just by randomly sampling the gaussian distribution
- The cost function pushes the codings to transform the latent space into a cloud of gaussian points.
- Loss has 2 terms : First is the binary cross entropy and the second is the latent loss term shown below. Latent loss term pushes the autoencoder to have codings that look as if they were sampled from a gaussian distribution

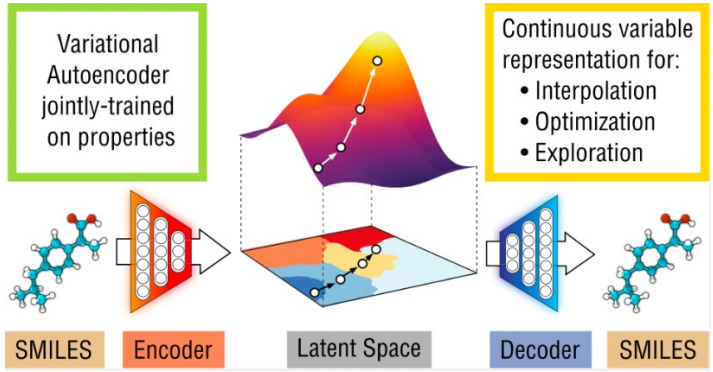
$$\mathscr{L} = -\frac{1}{2}\sum_{i=1}^{K} 1 + \log\left(\sigma_i{}^2\right) - \sigma_i{}^2 - \mu_i{}^2$$

- μ_i and σ_i are the mean and standard deviation of the ith component of the codings. KL divergence between target gaussian distribution and coding distribution.
- Rafeal et al. have used VAEs to discrete representations of molecules to and from a multi dimensional gaussian space.
- Encoder converts discrete representation into a continuous latent vector.
- Decoder converts continuous representations back into a discrete representation.
- Predictor operates in continuous latent space predicting chemical properties.
- Advantage of using continuous representations are that they automatically allow us to generate novel chemical structures by performing simple operations in the latent space such as perturbing known structures or interpolating between molecules. Allow use of gradient based optimization search tactics to find new functional compounds.



(a)

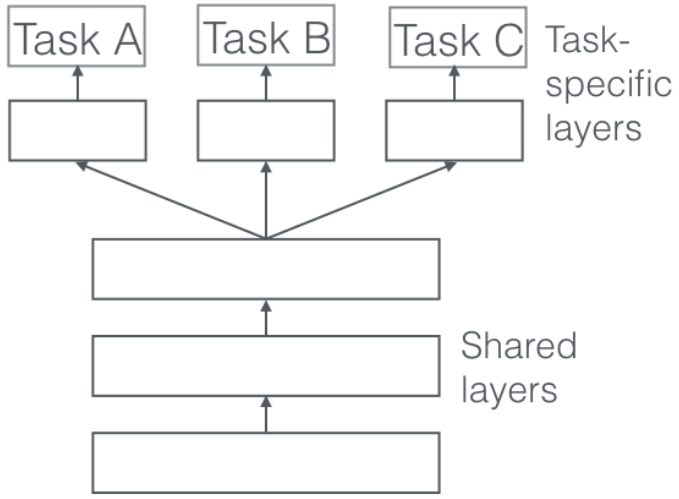




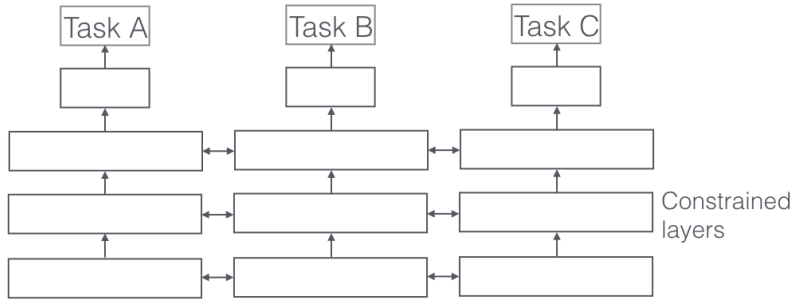
[1] Gómez-Bombarelli, R.; Wei, J. N.; Duvenaud, D.; Hernández-Lobato, J. M.; Sánchez-Lengeling, B.; Sheberla, D.; Aguilera-Iparraguirre, J.; Hirzel, T. D.; Adams, R. P.; Aspuru-Guzik, A. Automatic Chemical Design Using a Data-Driven Continuous Representation of Molecules. *ACS Cent. Sci.* **2018**, *4* (2), 268–276. https://doi.org/10.1021/acscentsci.7b00572.

Multi Task Learning (MTL) for Generalization

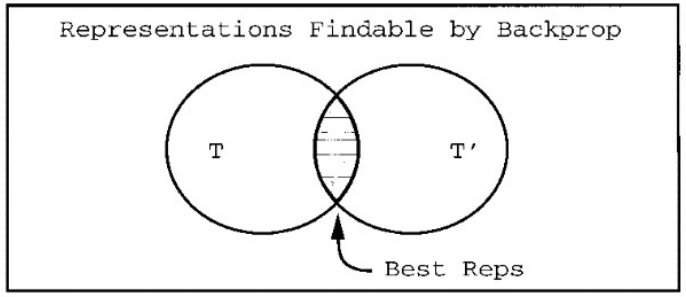
- Improves generalization performance by using a shared representation (latent space) for learning multiple tasks (example : learning how to classify (task 1) and reconstruct the input (task 2) accurately)
- Other names : joint learning, auxiliary task learning
- Optimizing over two loss functions. Intuitively the more the model has to optimize over multiple tasks, the lesser the chance it will overfit on a particular task.
- Generally MTL is done via two methods : Hard Parameter Sharing , Soft Parameter Sharing
  - Hard Parameter Sharing : Sharing hidden layers between all tasks but each task has its own task specific layer output.
  - Soft Parameter Sharing : Each task has separate model and separate parameters. Use regularization to ensure that each model task has similar parameters.
- Representation Bias : Consider two tasks (T and T').
  - T has local minima at A and B.
  - T' has local minima at A and C
  - When learning task T alone equally likely to find minima A or B. Similarly when learning task T' alone. Equally likely to find minima A or C. However joint learning task T and T' will drive the loss function to the minima A.
  - **MTL tasks prefer hidden layer representations that other tasks prefer.**
  - Now consider biasing T to fall into minima B instead of A.
  - Surprisingly T' falls into C. T' has no preference for A or C but falls into C due to the tide created when learning task T.
  - **MTL tasks prefer NOT to use hidden layer representations that other tasks prefer NOT to use.**



Hard Parameter Sharing



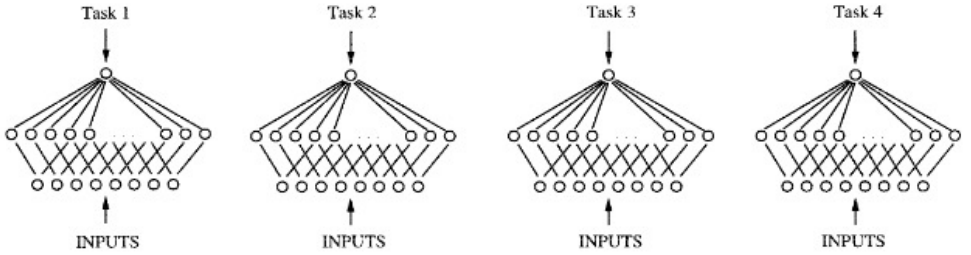
Soft Parameter Sharing



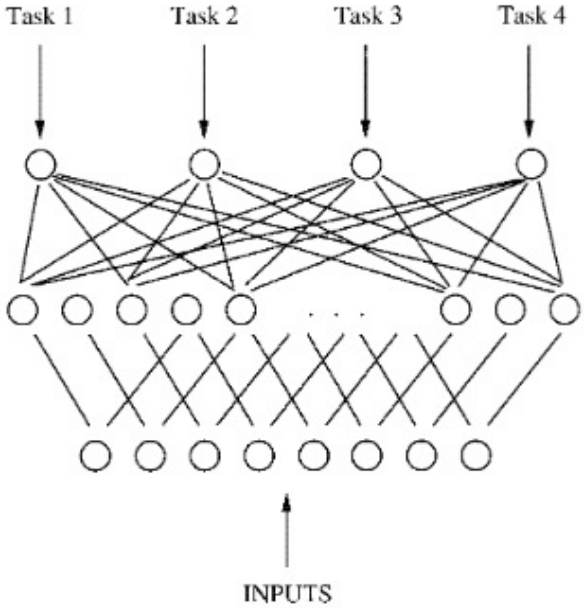
[1] https://ruder.io/multi-task/
[2] Caruana, R. Multitask Learning. 35.

Multi Task Learning (MTL) for Generalization

- When all outputs are connected to the same shared hidden layer each output will have its own weights and biases connected to the hidden layer and will also influence the weights and biases of lower layers through back prop.
- MTL improves generalization performance by inductive transfer
- How does MTL work ?
  - **Statistical Data Amplification** : Effective increase in sample size due to information contained in the training signals of related tasks
  - **Attribute Selection** : Two tasks training on T and T' that share same F will better select relevant attributes from input for constructing F.
  - **Eavesdropping** : Other tasks can eavesdrop on F constructed by a particular Task T allowing T'to be learnt better.
  - **Representation bias** : Since SGD is a stochastic search process, multiple runs will hardly ever yield the same result. In MTL search is biased towards representations that other tasks prefer.



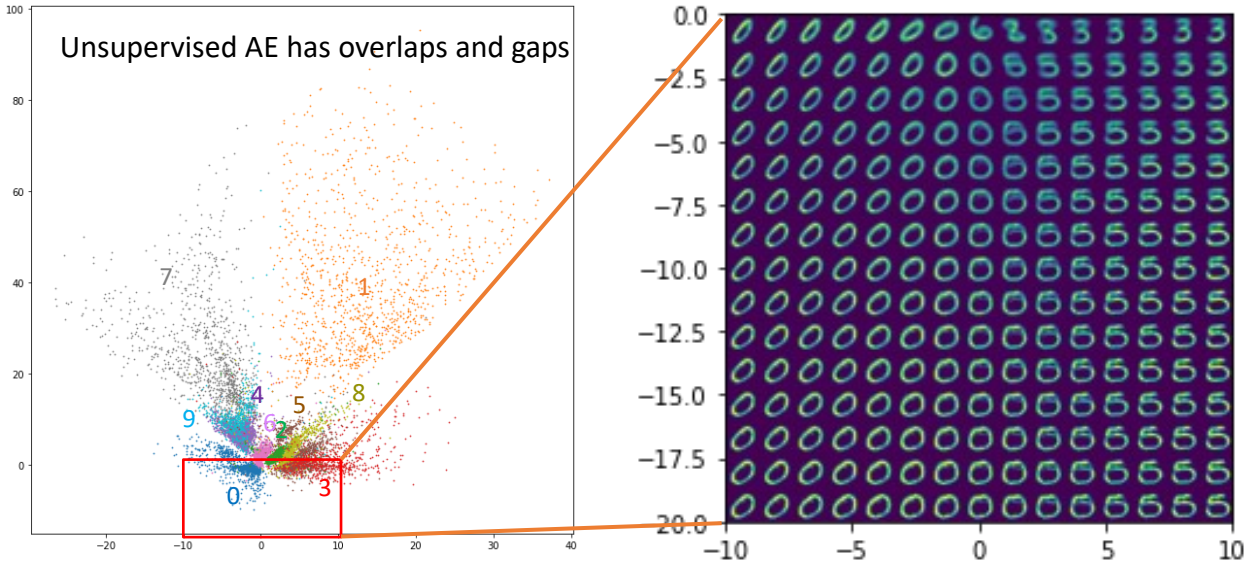
Single Task Learning (STL) – Each net is optimized in isolation. No net shares information with other nets



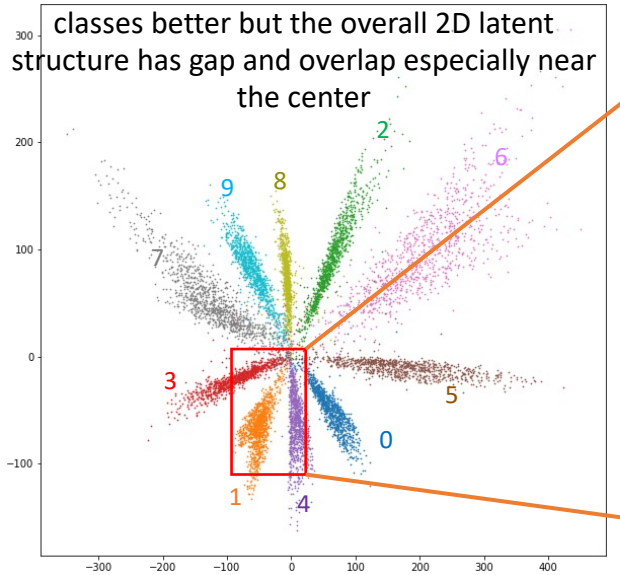
[1] https://ruder.io/multi-task/
[2] Caruana, R. Multitask Learning. 35.

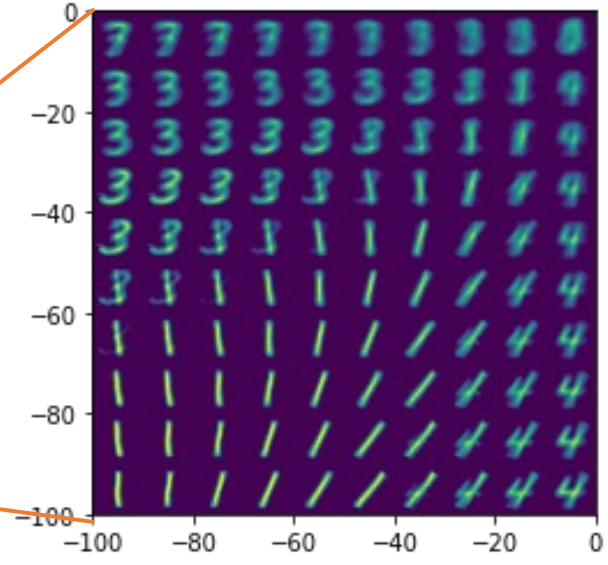
Deep generative models – Variational Autoencoders (VAE) [1][2]

- Traditional AE's simply learn to create a compressed encoding of an image by learning the proper weights and biases for the encoding and reconstruction process.
- They are also very good at clustering similar datapoints which is why a well separated latent space can be used to perform downstream classification tasks.
- Traditional AE's have a highly irregular latent structure which can be made tidy by adding supervision on the latent (Supervised Autoencoders). However, in both traditional and supervised AE's
  - Its not always clear what each dimension of the latent is encoding and hence interpolating along the dimension does not make sense.
  - The latent space is not suitable for generative processes due to gaps, overlap and broken class clusters (especially prevalent in traditional AE's) and the range of values taken by the latent variables is large.

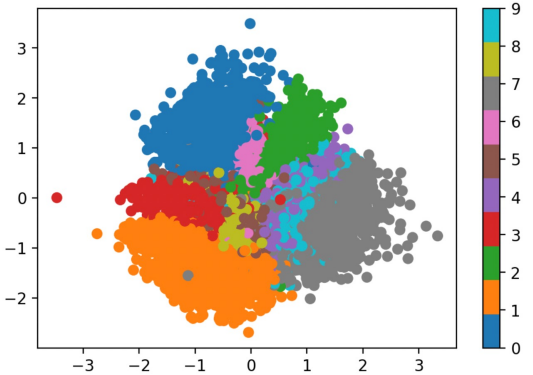


Supervised AE can segregate the different classes better but the overall 2D latent structure has gap and overlap especially near the center



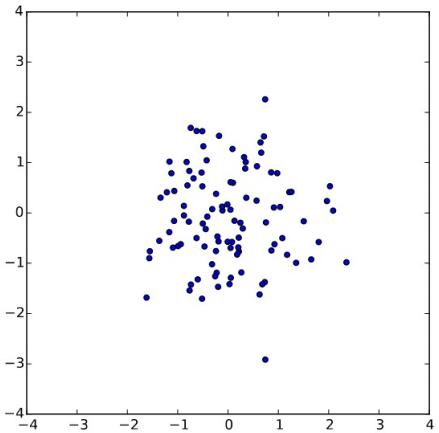


- Mathematical basis of VAE's has little to do with traditional autoencoders. Apart from the architecture that resembles like a traditional AE.
- VAE's attempt to map the distribution of data in high dimensions to a prior distribution in the latent space.
- **Called Variational since they use the variational inference principle**. From a set of densities over the latent we want to find the member from those densities that minimize the KL divergence to exact posterior. Each of these densities are defined by "variational parameters" -> The weights and biases of NN. Fitted density serves as proxy for the true density.
- **Similar to MCMC but MCMC samples a Markov chain to approximate the posterior while variational inference solves an optimization problem to arrive at an approximate posterior.**
- **MCMC more computationally expensive but we can get guarantees of sampling points form the exact distribution.**

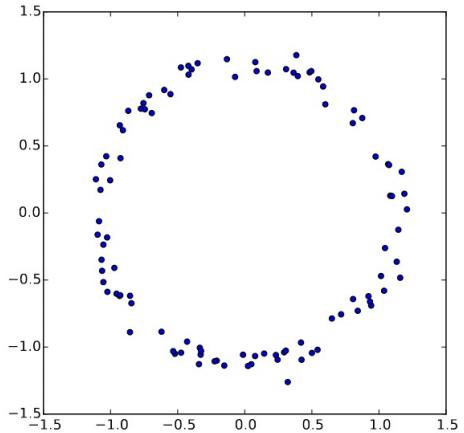


[1] Kingma, D. P., Welling, M. Auto-encoding variational bayes. arXiv preprint arXiv:1312.6114. (2013 May 1).
[2] Doersch, C. Tutorial on Variational Autoencoders. arXiv January 3, 2021. -- > Very Useful !
Other resources : 1) https://jaan.io/what-is-variational-autoencoder-vae-tutorial/ 2) https://keras.io/examples/generative/vae/ 3) https://avandekleut.github.io/vae/

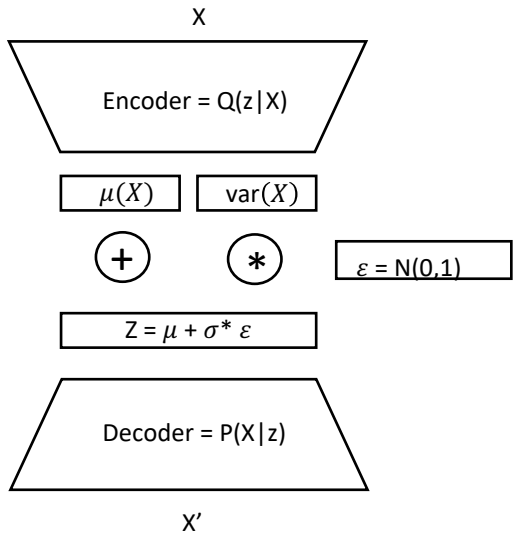
- Variational Inference is much faster because they can use stochastic optimization techniques. However, VI generally underestimates the variance of the posterior distribution.
- Let P(X) be the distribution of the input space, z is the latent space and P(z) is the distribution describing the latent space.
- Bayes theorem gives the theory on conditional probability P(X|z) = P(X,z)/P(z) or P(X,z) = P(X|z)P(z)
- From Total probability $\mathrm{P(X)} = \int P(X,z)dz = \int \mathrm{P(X|z)P(z)}dz$ where P(X|z) is the likelihood function and P(z) is the prior.
- Problem is we do not know what should be my latent variables 'z' and what the P(X|z) function is.
- Variational Autoencoders make this intractable problem tractable by selecting the latent variables the model finds useful and representative of X during the training (this is the job of the encoder) and then the decoder takes those latent variables and maps them to X.
- **The encoder is inferring** the latent variables that are representative of X and hence is modelling the posterior function Q(z|X).
- **The decoder is generating** the X from the chosen latent variables z and hence is modelling the likelihood function P(X|z).
- The weights and biases of the encoder and decoder NN are the parameters of the posterior and likelihood respectively.
- Analytically calculating the posterior or likelihood is intractable but VAE's can get very close to approximating them provided there is plenty of training data.
- The chosen prior to represent the latent can be any distribution as sufficiently sophisticated functions (posterior and likelihood) can transform variables from one distribution to another (see diagram below).



Samples from a
unit gaussian distribution



Samples passed through function
g(z) = z/10 + z/||z||



- Encoder in VAE, learns the posterior distribution Q(z|X) which is usually assumed to be Gaussian N($\mu(X)$, var(X))| X) . It takes in 'X' and outputs the mean and variance.
- The main idea behind VAE's is not to sample any random 'z' but only the ones that are likely to have produced 'X'. The space of the 'z' values that are likely under Q(z|X) should be smaller than the space of all z's that are likely under the prior P(z).
- Decoder in VAE learns the likelihood distribution i.e p(X|z). It takes in a sampled 'z' and outputs the reconstruction X.

Setting up and Optimizing the Objective

Kullback-Leibler Divergence (KL Divergence or D)

↳ Some arbitrary Q which may or may not depend on X.

$$D[Q(z)\,||\,P(z|X)] = E_{z\sim Q}[\log Q(z) - \log P(z|X)]$$

$$D_{KL}[P||Q] = \sum P(x)\log\left(P(x)/Q(x)\right)$$

We can bring in $P(X|z)$ and $P(X)$ using Bayes rule. $\left[P(X|z)P(z) = P(z|X)P(X)\right]$

$$D[Q(z)\,||\,P(z|X)] = E_{z\sim Q}[\log Q(z) - \log(P(X|z)) - \log P(z) + \log P(X)]$$

$$D[Q(z)||P(z|X)] = E_{z\sim Q}[\log Q(z) - \log P(X|z) - \log P(z)] + \log P(X)$$ (Taking it out since it does not depend on z)

$$\log P(X) - D[Q(z)||P(z|X)] = E_{z\sim Q}[\log P(X|z)] - D[Q(z)||P(z)]$$

Since we interested in inferring a $P(X)$ it makes sense to construct a 'Q' that does depend on X.

$$\underbrace{\log P(X) - D[Q(z|X)||P(z|X)]}_{\text{Quantity we want to maximize}} = \underbrace{E_{z\sim Q}[\log P(X|z)]}_{\text{Term 1}} - \underbrace{D[Q(z|X)||P(z)]}_{\text{Term 2}}$$ maximize the ELBO

Can optimize via Gradient Descent

- KL term on left hand side is pulling $Q(z|X)$ to match $P(z|X)$. If we use a high capacity model (appr. neural network architecture) then we can hopefully match $P(z|X)$ and have made an intractable distribution tractable.
- Next lets say that $Q(z|X)$ takes the form of $N(z|\mu(X;\nu), \Sigma(X;\nu))$ where μ and Σ are deterministic functions that have parameters 'ν'.
- μ and Σ are implemented via NN. Σ is constrained to be a diagonal matrix.

↳ $Ce^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$

Term 2

$$-D[N(\mu_0,\Sigma_0)||N(\mu_1,\Sigma_1)] = \frac{1}{2}\left[\text{tr}(\Sigma_1^{-1}\Sigma_0) + (\mu_1-\mu_0)^T\Sigma_1^{-1}(\mu_1-\mu_0) - k + \log\left(\frac{\det\Sigma_1}{\det\Sigma_0}\right)\right]$$

For our prior $P(z) = N(0, I)$ this simplifies to

$$-D[N(\mu_0,\Sigma_0)||N(0,I)] = \frac{1}{2}\left[\text{tr}(\Sigma(X)) + \mu(X)^T\mu(X) - k - \log\det(\Sigma(X))\right]$$

This shows that we want our latents to be independent

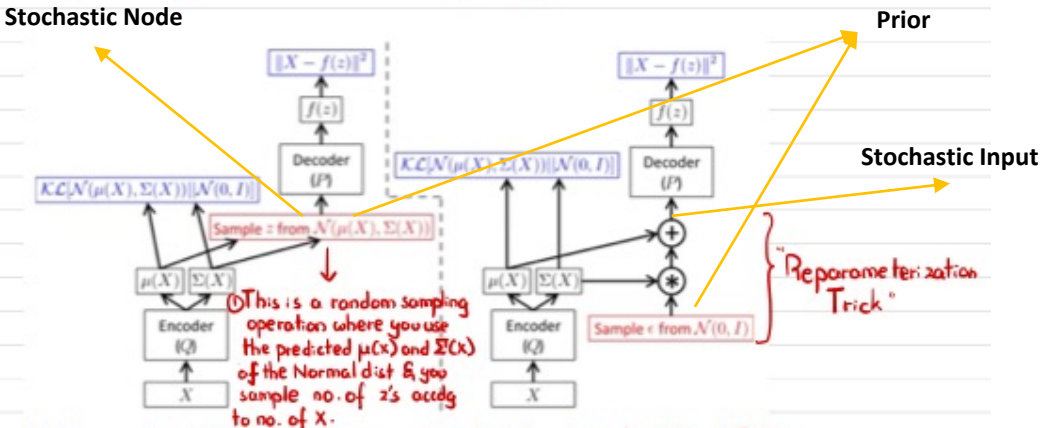
Term 1 We could calculate $E_{z\sim Q}[\log P(X|z)]$ when doing batch or mini batch gradient descent. In SGD we just use $\log P(X|z)$ as an approximation of $E_{z\sim Q}$

We are doing SGD for different X in dataset D. Therefore equation becomes

$$E_{X\sim D}[\log P(X) - D[Q(z|X)||P(z|X)]] = E_{X\sim D}\left[\underbrace{E_{z\sim Q}[\log P(X|z)]}_{(1X)} - D[Q(z|X)||P(z)]\right]$$

Gradient can be moved into $E_{X\sim D}$. If we are using one training example and sampling 1 latent vector z to represent that X. We can compute gradient of $\log P(X|z) - D[Q(z|X)||P(z)]$ We then average this gradient over arbitrarily large X.

However an important problem arises when back propagating the errors :



① This is a random sampling operation where you use the predicted $\mu(X)$ and $\Sigma(X)$ of the Normal dist & you sample no. of z's accdg to no. of X.

② However doing it this way there is no way for BackProp to reach $\mu(X)$ and $\Sigma(X)$ for value adjustment. BackProp can work with stochastic inputs as shown in right diagram but not stochastic nodes.

$$\log P(X|z) = \log P(X|z = \mu(X) + \Sigma^{1/2}(X) * \epsilon)$$

Reparameterization can only work if $Q(z|X)$ must be continuous.

Interpreting the Objective

- The tractability of $P(z|X)$ relies on the assumption that X can be modelled as some Gaussian distribution with mean $\mu(X)$ and variance $\Sigma(X)$
- $P(X)$ converges only if $D[Q(z|X)||P(z|X)] \longrightarrow 0$.
- $D[Q(z|X)||P(z)]$ can be viewed as the extra information we get about X when z comes from $q(z|X)$ rather than from $P(z)$.
- $-\log P(X)$ can be seen as the total number of bits required to construct X using an ideal encoding

Derivation of KL loss for Guassian Prior (VAE)

Derivation of KL loss for Gaussian Prior

Variational lower bound = $E[P(X|z)] - KL(Q(z|X)||P(z))$

Goal is to maximize the variational lower bound

Reconstruction Error (this is L2 loss if we assume $P(X|z)$ is gaussian)

- Information $\propto \frac{1}{p(E)}$

Information content in highly probable event is low
Information content in low prob. event is high.

$$\log(p(x)) \propto p(x)$$

$$\log\left(\frac{1}{p(x)}\right) \propto \frac{1}{p(x)}$$

$$-\log(p(x)) \propto \frac{1}{p(x)}$$

I

Information content of event X wrt p
$I_p(x) = -\log(p(x))$
Information content of event X wrt q
$I_q(x) = -\log(q(x))$

These are 2 distributions

- Difference in information b/w $q(x)$ and $p(x)$ is

$$\Delta I = I_q(x) - I_p(x) = -\log(q(x)) + \log(p(x))$$

$$\Delta I = \log\left(\frac{p(x)}{q(x)}\right)$$

Expected value for cont. RV (i.e.) $E[X] = \int x f(x) dx$

- Kullback-Leibler divergence is expectation of the above difference

"Expectation of ΔI wrt $p(x)$" $D_{KL}(p(x)||q(x)) = E_{x\sim p}[\Delta I] = \int p(x) \log\left(\frac{p(x)}{q(x)}\right) dx$

"Expectation of ΔI wrt $q(x)$" $D_{KL}(q(x)||p(x)) = E_{x\sim q}[\Delta I] = \int q(x) \log\left(\frac{q(x)}{p(x)}\right) dx$

$D_{KL}(p(x)||q(x)) \neq D_{KL}(q(x)||p(x))$ (Not Symmetric)

- For the above reason Kullback-Leibler is called Divergence and not a metric.

$$D_{KL}(q(x)||p(x)) = -\int q(x) \log\frac{q(x)}{p(x)} dx \geq 0 \qquad \log t \leq t - 1$$

Our approximation to True Posterior parameterized by θ

$$D_{KL}(q_\theta(z|x_i)||p(z|x_i)) = \int q_\theta(z|x_i) \log\frac{q_\theta(z|x_i)}{p(z|x_i)}$$

True Posterior

$p(z|x) = \frac{p(z,x)}{p(x)}$

$$= \int q_\theta(z|x_i) \log\frac{q_\theta(z|x_i)\,p(x)}{p(x|z)\,p(z)}$$

$p(z|x)p(x) = p(x|z)p(z)$
$p(z|x) = \frac{p(x|z)p(z)}{p(x)}$

$$= \int q_\theta(z|x_i)\left[\log(q_\theta(z|x_i)) + \log p(x) - \log p(x|z) - \log p(z)\right]$$

$$= \log p(x) - E_{q_\theta}[\log p(x|z)] + E_{q_\theta}\left[\log\left(\frac{q_\theta(z|x_i)}{p(z)}\right)\right]$$

We know $D_{KL}(q_\theta(z|x_i)||p(z|x_i)) \geq 0$

$$\log p(x_i) \geq E_{q_\theta}[\log p(x_i|z)] - E_{q_\theta}\left[\log\left(\frac{q_\theta(z|x_i)}{p(z)}\right)\right]$$

$$\log p(x_i) \geq E_{q_\theta}[\log p(x_i|z)] - D_{KL}(q_\theta(z|x_i)||p(z))$$

Evidence lower Bound (ELBO)
or
Variational Lower Bound

- Maximizing our ELBO, maximizes the log probability of our data by proxy
- This is core idea of variational inference, since maximization of log prob. directly is intractable
- We do not know the true distribution over z looks like. We can have different priors such as Multivariate Gaussian, Mixture of Gaussians, Von Mises distribution etc. reflecting our multiple different beliefs of how the latent is distributed.
- The task of the encoder is to learn the most optimal $q_\theta(z|x_i)$ with the hope it reflects the actual $p(z|x_i)$. This is all based on the assumption that there exists some latent variables (z) that can explain the relations existing within the observed data 'x'.

$z_i \in \mathbb{R}^m$ and $x_i \in \mathbb{R}^n$ $(m<n)$

$p(z|x_i)$ tells us given an x_i what is the probability of observing $z_i \in z$

↳ We want to get an approximation that is as similar as possible to $p(z|x_i)$ in order to accurately reflect our belief $p(z)$

We choose the one that maximizes our $p(x)$ the most.

$X = \{x_1, x_2, x_3, x_4\}$ Set of 4 features $\qquad P(X) = P(x_1, x_2, x_3, x_4)$ (joint probability distribution)

* Complex distribution that holds the key information of how high D points are arranged.

- Can calculate marginal distributions for each feature as follows

$$P(x_1) = \int x_2 \int x_3 \int x_4 P(x_1, x_2, x_3, x_4)\, dx_4\, dx_3\, dx_2$$

- Now what if there is a hidden latent variable z_i we could project each x_i onto. This hidden variable z_i would be structured to capture the essential patterns in the observed data.
- We first start with a belif of how these points are clustered denoted by $P(z_i)$
- We then learn the most optimal $q_\theta(z|x_i)$ to project each x_i onto 'z'.
- Simultaneously our decoder learns $q_\phi(x_i|z_i)$ such that we can maximize our evidence lower bound (ELBO).

Key concept is introduction of Reparameterization technique.

- Allows errors to back propogate to encoder to learn optimal $q_\theta(z|x_i)$
- Need to sample from $p(z_i)$ to send to decoder
- Also need to update $q_\theta(z_i|x_i)$ to reach optimal $q_{\theta^*}(z|x_i)$

$$z = \begin{bmatrix}\mu_1 & \mu_2 & \mu_3\end{bmatrix} + \begin{bmatrix}\sigma_1 & \sigma_2 & \sigma_3\end{bmatrix} * \epsilon$$

↳ sampled from $N(0,1)$

Now it actually looks like it is sampled from $N(0,1)$

$X = \{x_1, x_2, x_3 \dots x_N\}$

$P_{\theta_1}(X) = P_{\theta_1}(x_1, x_2 \dots x_N)$

Assuming $x_1, x_2 \dots x_N$ are independent of each other

$P_{\theta_1}(X) = P_{\theta_1}(x_1) P_{\theta_1}(x_2) P_{\theta_1}(x_3) \dots P_{\theta_1}(x_N)$

$\log P_{\theta_1}(X) = \log P_{\theta_1}(x_1) + \log P_{\theta_1}(x_2) \dots \log P_{\theta_1}(x_N)$

$\log P_{\theta_1}(X) = \sum_{i=1}^{N} \log P_{\theta_1}(x_i) \rightarrow$ log liklihood

obj. max $\log P_{\theta_1}(X)$ Maximize log liklihood

Method 2: Find 'z' that are representative of X
Use the 'z' to reconstruct X.

$P(z|X) = \frac{P(z,X)}{P(X)}$

$P(z|X)P(X) = P(X|z)P(z)$

$P(z|X) = \frac{P(X|z)P(z)}{P(X)}$

Mathematically the distribution we want is $P(z|X)$
We approximate $P(z|X)$ with $Q(z|X)$

We measure similarity between $P(z|X)$ and $Q(z|X)$ using KLD

$$D(Q(z|X)||P(z|X)) = \int Q(z|X) \log \frac{P(z|X)}{Q(z|X)} = E_Q[\log P(z|X) - \log Q(z|X)]$$

$$= E_Q[\log P(X|z) + \log P(z) - \log P(X) - \log Q(z|X)]$$

$$D(Q(z|X)||P(z|X)) + \log P(X) = E_Q[\log P(X|z)] - D[Q(z|X)||P(z)]$$

$$\log P(X) \geq E_Q[\log P(X|z)] - D[Q(z|X)||P(z)]$$

= only when $D[Q(z|X)||P(z|X)] = 0$

Maximize variational lowerbound on liklihood function or maximize the ELBO

Integral substituted with Σ

$$P(X|z) = C \times \exp\left(-\frac{(X - f_{\theta_1}(z))^2}{2\sigma^2}\right)$$

$$\log P(X|z) = -(X - f_{\theta_1}(z))^2 \quad \text{L2 loss}$$

Extending VAE's

1. Using different posterior and prior distributions [1][2][3]

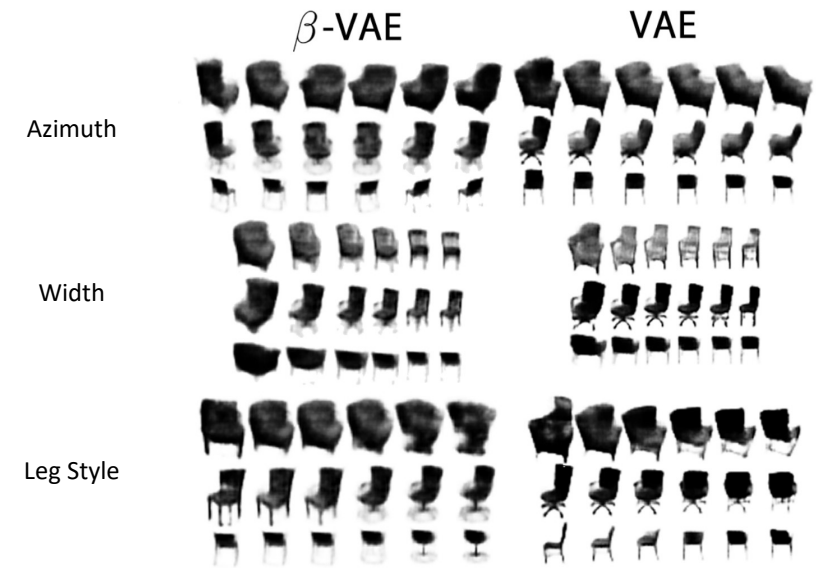
- In some cases the underlying factor that generated the dataset can have a certain geometric or topological structure that cannot be captured with the typical hyperplane assumption. **This is called manifold mismatch.**
- For example in the Cars3D dataset one factor of variation is the azimuthal angle of rotation of the car. The geometrical structure of this feature is circular and hence a periodical latent variable can represent this feature more appropriately.
- Using von Mises Fisher distribution also called as wrapped normal or circular normal distribution and using a uniform prior allows data points to spread out on the hypersphere rather than imposing a constraint for them to cluster around the origin as with having a unit gaussian prior.
- Other priors include mixture of gaussians

2. Normalizing Flows [4]

- They are a series of invertible transformations applied sequentially to an initial reparameterizable density, allowing for more complex posteriors. While allowing for a more flexible posterior, they do not modify the standard normal prior assumption.

3. Learning Disentangled Representations [5]

- Main idea is to **learn an independent factorized representation of the independent data generative features**.
- A disentangled representation can be defined as one where single latent units are sensitive to changes in single generative factors, while being relatively invariant to changes in other factors.
- For example, a model trained on a dataset of 3D objects might learn independent latent units sensitive to single independent data generative factors, such as object identity, position, scale, lighting or colour, thus acting as an inverse graphics model.
- β-VAE is a deep unsupervised generative approach for disentangled factor learning that can **automatically discover the independent latent factors** of variation in unsupervised data.
- β-VAE with β = 1 corresponds to the original VAE framework.



VAE always learns an entangled representation (Chair width is always entangled with azimuth and leg style)

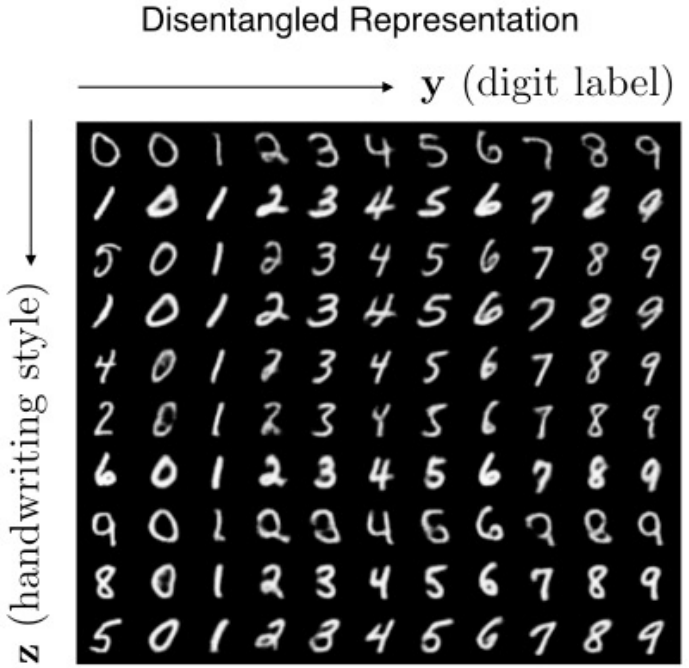
[1] Falorsi, L.; de Haan, P.; Davidson, T. R.; De Cao, N.; Weiler, M.; Forré, P.; Cohen, T. S. Explorations in Homeomorphic Variational Auto-Encoding. arXiv July 12, 2018.
[2] Davidson, T. R.; Falorsi, L.; De Cao, N.; Kipf, T.; Tomczak, J. M. Hyperspherical Variational Auto-Encoders. arXiv September 26, 2018.
[3] Perez Rey, L. A.; Menkovski, V.; Portegies, J. Diffusion Variational Autoencoders. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*; International Joint Conferences on Artificial Intelligence Organization: Yokohama, Japan, 2020; pp 2704–2710. https://doi.org/10.24963/ijcai.2020/375.
[4] Rezende, D. J.; Mohamed, S. Variational Inference with Normalizing Flows. arXiv June 14, 2016.
[5] Higgins, I.; Matthey, L.; Pal, A.; Burgess, C.; Glorot, X.; Botvinick, M.; Mohamed, S.; Lerchner, A. β-VAE: LEARNING BASIC VISUAL CONCEPTS WITH A CONSTRAINED VARIATIONAL FRAMEWORK. **2017**, 22.

- The extra pressures coming from high β values, however, may create a trade-off between reconstruction fidelity and the quality of disentanglement within the learned latent representations. Disentangled representations emerge when the right balance is found between information preservation (reconstruction cost as regularisation) and latent channel capacity restriction (β > 1).

$$\mathcal{F}(\theta, \phi, \beta; \mathbf{x}, \mathbf{z}) \geq \mathcal{L}(\theta, \phi; \mathbf{x}, \mathbf{z}, \beta) = \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})] - \beta\, D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})||p(\mathbf{z}))$$

4. Specifying latent variables for the model to use [1]

- We can leave it up to the model to determine what should each dimension of the latent contain using β-VAE's. However we can also restrict the interpretability of certain dimensions which contains our apriori knowledge of the domain that the network must learn to capture while leaving the rest to be learned in an entangled/disentangled manner.
- An example is In case of MNIST having an explicit 'digit' latent variable gives a constant axis of variation and the rest of the other latent dimensions can be used to represent different styles of writing the MNIST digit. 'y' captures the classification label and 'z' captures all other implicit features such as pen type and handwriting style.

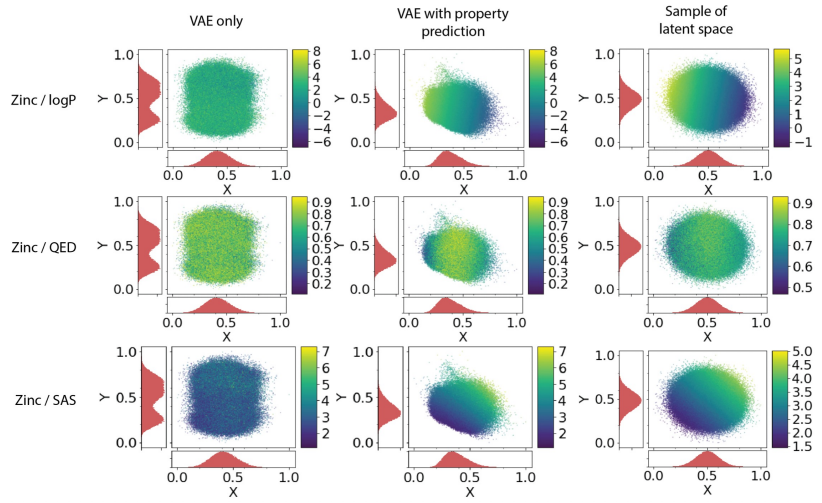


[1] Siddharth, N.; Paige, B.; van de Meent, J.-W.; Desmaison, A.; Goodman, N. D.; Kohli, P.; Wood, F.; Torr, P. H. S. Learning Disentangled Representations with Semi-Supervised Deep Generative Models. arXiv November 13, 2017.
[2] Kulkarni, T. D.; Whitney, W.; Kohli, P.; Tenenbaum, J. B. Deep Convolutional Inverse Graphics Network. arXiv June 21, 2015.

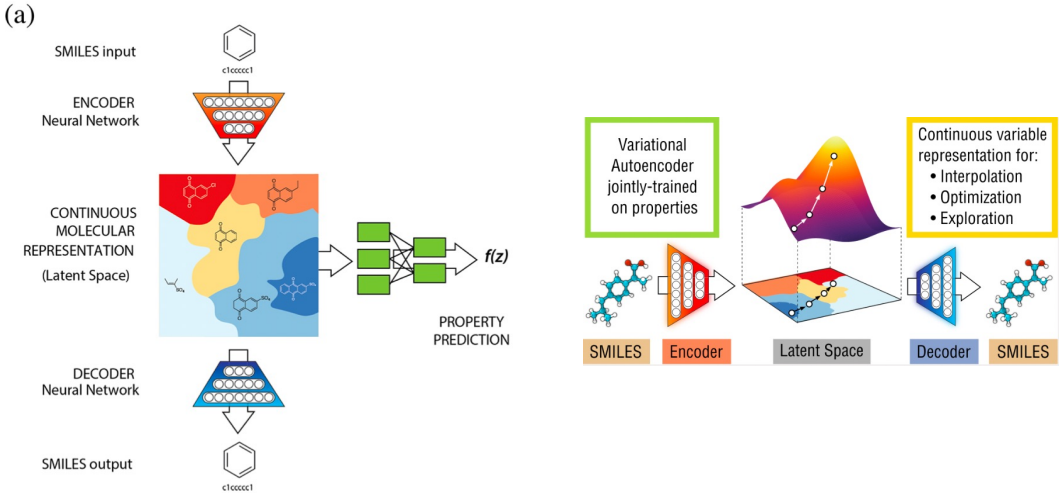
Variational Autoencoder with property prediction attached to latent

- One famous recent paper published in 2017 that does this is by the Alan Aspuru Guzik Group for continuous representation of molecules. [1]
- **Model architecture :**
  - Encoder used 3 1D conv layers followed by one fully connected layer of width 196 (latent). Decoder used recurrent neural network layers. Property prediction networks were two fully connected layers with 1000 nodes. Property prediction done on z_mean latent vector.
  - Regression network trained along with autoencoder for property prediction. Properties predicted include logP (water-octanal partition coefficient), QED (Qualitative Estimate Drug-likeness), SAS (Synthetic Accessibility Score), HOMO, LUMO and electronic spatial extent.
- **Continuous encoding of structures of molecules :**
  - Molecules represented as SMILES strings are converted to a cont. vector representation. Strings of characters can be encoded into vectors using recurrent neural networks (RNNs). Convolutional AE's (use 1D conv nets) can also be used for string encoding because of repetitive, translationally invariant substrings that correspond to chemical substructures (Ex : cycles and functional groups)
- **Interpolation in high D spaces :**
  - Continuous latent space allows interpolation of molecules. However when interpolating in high dimensional space, linear interpolation or Euclidean distance does not map directly to similarity between molecules. This is because most of the mass of independent distributed normal random variables lies not near the mean but in an annulus around it. Similar to the idea that most of the volume of a high dimensional orange is in the skin and not in the pulp. Hence when interpolating must use spherical interpolation (slerp)

[1] Gómez-Bombarelli, R.; Wei, J. N.; Duvenaud, D.; Hernández-Lobato, J. M.; Sánchez-Lengeling, B.; Sheberla, D.; Aguilera-Iparraguirre, J.; Hirzel, T. D.; Adams, R. P.; Aspuru-Guzik, A. Automatic Chemical Design Using a Data-Driven Continuous Representation of Molecules. *ACS Cent. Sci.* **2018**, *4* (2), 268–276. https://doi.org/10.1021/acscentsci.7b00572.



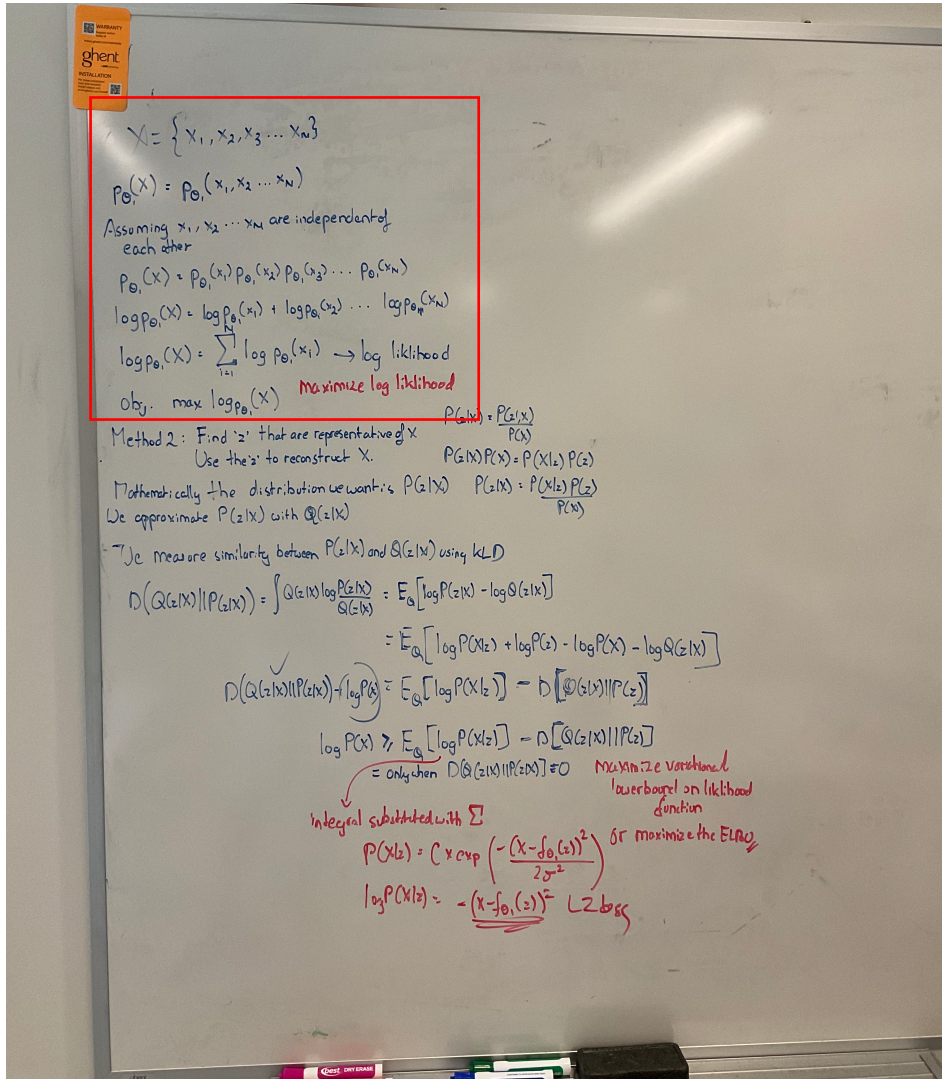
Adding 'y' to latent helps segregate latent based on property values. Graphs plotted using PCA



A paper which uses the sampled z for classification :
Ryu, K. H.; Batbaatar, E. Improved Cancer Classification with Supervised Variational Autoencoder on DNA Methylation Data. In *Advances in Intelligent Information Hiding and Multimedia Signal Processing*; Pan, J.-S., Li, J., Ryu, K. H., Meng, Z., Klasnja-Milicevic, A., Eds.; Smart Innovation, Systems and Technologies; Springer Singapore: Singapore, 2021; Vol. 212, pp 36–43. https://doi.org/10.1007/978-981-33-6757-9_5.

A paper which uses the z_mean and z_log_var for classification :(1)
Ji, T.; Vuppala, S. T.; Chowdhary, G.; Driggs-Campbell, K. Multi-Modal Anomaly Detection for Unstructured and Uncertain Environments. arXiv December 15, 2020.

- Suppose we want our model to generate dog images. X is the set of 128x128 grey scale dog images. If we could figure what pixels should light up and by how much then we can generate new dog images.
- Traditional autoencoders are not generative models and simply learn to reconstruct the input from a low dimensional code.
- What we want is to find the distribution of dog images (X) and sample 128*128 pixel values from that distribution to generate a new dog image.
- The problem is what does this distribution look like? How can we model such high dimensional distributions ?
- Mathematically the objective we are trying to maximize is as follows :



- We would ideally want a low dimensional distribution and is something we are more familiar with. Say this distribution is defined by variable 'z' and we assume it to be a unit gaussian. This becomes our prior. Or our assumption of what the low dimensional distribution should look like.
- Now the task at hand is how do we map points in the high dimensional distribution defined by X to this low dimensional prior. We use the help of another distribution called the posterior P(z|X). Let assume that the posterior is also gaussian for simplicity.
- We leave it up to the encoder to model this posterior distribution.
- Now using the 'z' output by the encoder the task now is to use these smapled 'z' to generate X. That is done by the doder which is also a neural net and it model the P(X|z) or the likelihood which we also assume to be gaussian. (Reason for L2 loss)
- The total loss function that is optimized is derived in the picture.

$$L_{VAE} = E_{Q(z|X)}[\log(P(X|z)] - D_{KL}(Q(z|X)||P(z))$$

$$P(X|z) = N(X; f(z), \sigma^2 I) = C * \exp\left(\frac{-(X - f(z))^2}{2\sigma^2}\right)$$

$$\log(P(X|z)) = \sum_{i=1}^{N} logC * \exp\left(\frac{-(x^i - f(z^i))^2}{2\sigma^2}\right) = NC - \frac{1}{2\sigma^2}\sum_{i=1}^{N}(x^i - f(z^i))^2$$

$$L_{VAE} = L_{l2} + L_{prior}$$

Representation Learning

- Wasserstein Autoencoders : https://arxiv.org/pdf/1711.01558.pdf
- Sliced Wasserstein Autoencoders : https://arxiv.org/pdf/1804.01947.pdf
- Idea is to use the Wasserstein metric to pose penality on the distance between encoded and prior distribution instead of using KL divergence.
- In probability theory, f-divergence measures the difference between two probability distributions P and Q. Common divergences such as KL-divergence, Hellinger distance, Jensen-Shannon divergence are special cases of f-divergence.
- Drawbacks with KL –Divergence :
  - Not symmetric $D_{KL}(P||Q)$ is not equal to $D_{KL}(Q||P)$
- Optimal transport is another way to measure distance between probability distributions. Distance is symmetric and supports the triangle inequality.
- Intuition behind optimal transport :
  - Consider dirt piles and holes and the idea is to move dirt into the holes in the most efficient way as possible. The volume of dirt in the piles and the volume of the holes is the same. Consider the dirt piles and holes to be probability distributions P and Q respectively.
  - How do we know which is the most efficient transportation plan. For that we use a transporation cost C(x0,y0,x1,y1) which is the cost of moving I unit of dirt from cood (x0,y0) to (x1,y1). This is given by the euclidean distance.
  - Read this paper here : http://alexhwilliams.info/itsneuronalblog/2020/10/09/optimal-transport/

Conditioning the latent as a Guassian Mixture Model

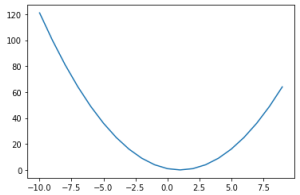
- Assumption : Observed data is generated from a multi model prior distribution.
- Problem with traditional VAE's is the concern of over regularization which is mitigated by using the Minimum Information Constraint (MIC).
- Amortized inference ?
- Check structured VAE's.
- Stacked VAE's implemented in Semi Supervised Learning with Deep Generative Models
- Adverserial Autoencoders
- CatGANs
- Also check out Principle Component Analysis Autoencoders
- https://arxiv.org/pdf/1611.02648.pdf
- https://arxiv.org/pdf/1406.5298.pdf
- Generative Adversarial, Variational Autoencoders, Flow models, Autoregressive models are the current top deep generative models.
- Code for a simple Gaussian Posterior and Prior on the latent : https://github.com/aspuru-guzik-group/chemical_vae/blob/main/chemvae/models.py

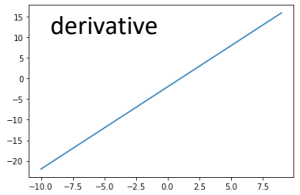
Reconstruction Losses in Autoencoders

- The most typically used loss function is the L2 loss.
- In autoencoders we have input x, latent z. z is sampled from the distribution q(z|x) (posterior) and the reconstruction $\hat{x}$ is sampled from the distribution p(x|z).
- In variational AE there are two crucial important loss functions. One is the reconstruction loss gaussian distribution (prior). function and the other is the loss function on the latent. By introducing a loss function on the latent we are shaping it to take a particular form. The most widely used form is of course the gaussian distribution.
- Typically we assume p(x|z) = N($\hat{x}, \sigma$) $\propto$ exp(-||x - $\hat{x}$||^2) and that is where the L2 loss function comes from. L2 scales with error and hence penalizes large deviations more than compared to small deviations. Optimal prediction is the mean. Problem with L2 loss is that it i) is sensitive to large noise in the image ii) can lead to blurry image reconstructions since it does not penalize small reconstruction errors as harshly as big ones.
- If we assume p(x|z) $\propto$ exp(-||x - $\hat{x}$||) then we get the L1 loss function. L1 loss acts as a regularizer and helps introduce sparsity in images giving more detailed images rather than the blurry images as produced using L2. L1 out performs L2 when used in image restoration tasks such as de noising and de blurring. Optimal prediction is the median. Problem with L1 is that whatever be the error size it penalizes all equally since the derivative is a constant and only the sign changes which is advantageous compared to L2 since it also penalizes small errors but there can be convergence issues for error values close to 0 due to oscillations about +/- 1.
- L2 loss does not penalize small reconstruction errors much as L1 does. In case of VAE the total loss is then dominated by the second term which is the KL divergence. Increasing the reconstruction loss weight more hampers the construction of the latent space when its error is large. The idea is then to increase the weight of the reconstruction error when the reconstruction error is small but to clip the increase in the reconstruction error so that it does not increase linearly. A good approximation to that is the tanh function whose integral is the logcosh function.
- Another similar loss function to logcosh is the Huber loss function which is a piece wise defined loss function

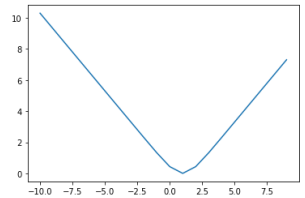
$$f(t;a) = \frac{1}{a}\log(\cosh(at)) = \frac{1}{a}\log\frac{e^{at}+e^{-at}}{2} \rightarrow \begin{cases} |t| - \frac{1}{a}\log 2, & \text{when } |t| \rightarrow \infty \\ 0.5at^2, & \text{when } |t| \rightarrow 0 \end{cases}$$

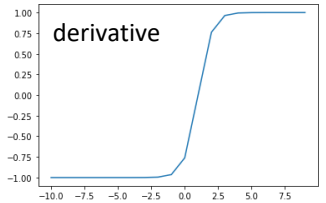
$$L2 = \sum_i^N (x_i - \hat{x_i})\text{^}2$$



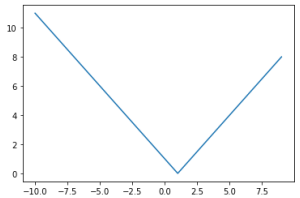


$$L_{logcosh} = \frac{1}{a}\sum_i^N \log(\cosh(a(x_i - \hat{x_i})$$

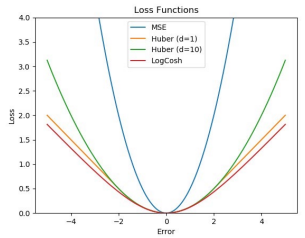


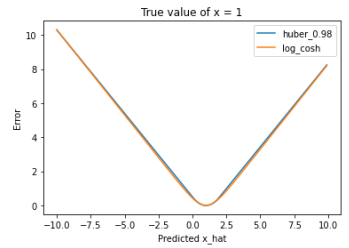


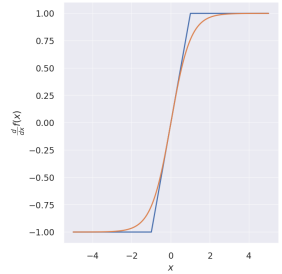
$$L1 = \sum_i^N (x_i - \hat{x_i})$$



$$\text{Huber}(x) = \begin{cases} \frac{1}{2}x^2 & \text{for } |x| \le \delta, \\ \delta|x| - \frac{1}{2}\delta^2 & \text{otherwise.} \end{cases}$$







Required features for the data driven model

1. Extract the **relevant set of synthesis parameters (L)** from the **full set of synthesis parameters (X) (Feature Extraction, Dimensionality Reduction)**

$$X = \{x_1, x_2, x_3, x_4, \dots x_n\} \in \mathbb{R}^n$$

2. Physical interpretation of the **latent variables (Model interpretability)**

$$L = \{l_1, l_2, l_3, l_4, \dots l_m\} \in \mathbb{R}^m$$

3. Learn how the **latent variables are related to property/properties at each scale (Classification/Regression)**

$$y = \boldsymbol{f}(l_1, l_2, l_3, l_4, \dots l_m)$$

4. Uncertainty quantification **(Model interpretability)**

$$\hat{y} = \hat{f}(l_1, l_2, l_3, l_4, \dots l_m) + \sigma_{noise}$$

epistemic uncertainity

aleatory uncertainity

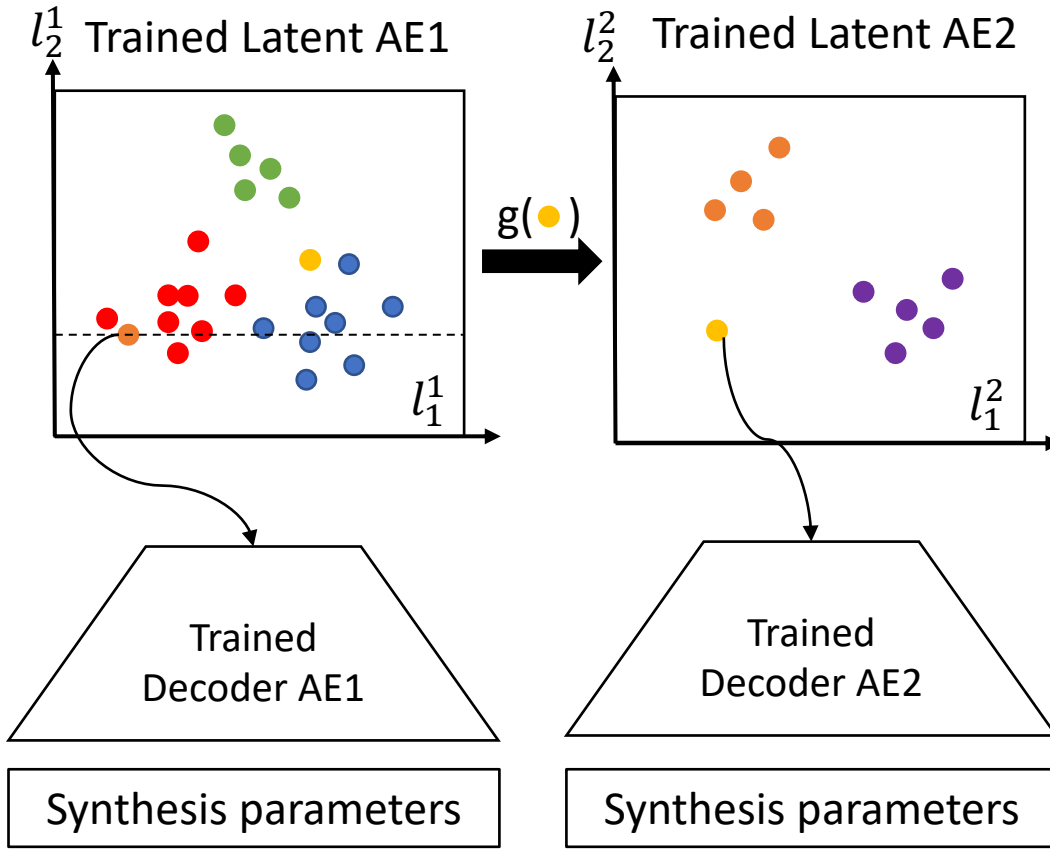
Computing Prediction intervals : Lai, Y.; Shi, Y.; Han, Y.; Shao, Y.; Qi, M.; Li, B. Exploring Uncertainty in Deep Learning for Construction of Prediction Intervals. arXiv April 26, 2021.

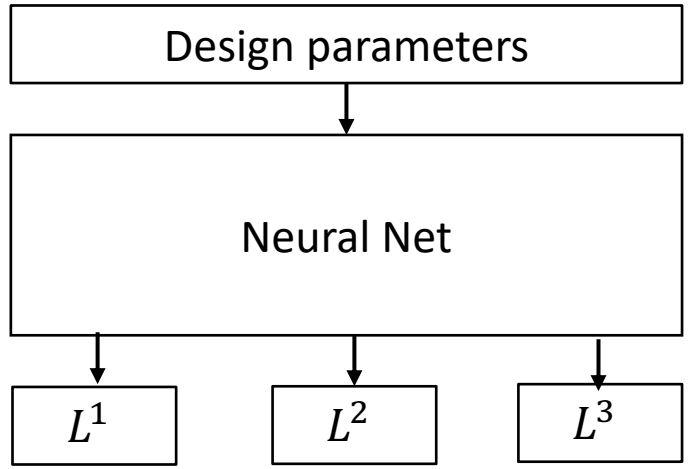
Bayesian Neural Nets : Jospin, L. V.; Buntine, W.; Boussaid, F.; Laga, H.; Bennamoun, M. Hands-on Bayesian Neural Networks -- a Tutorial for Deep Learning Users. *IEEE Comput. Intell. Mag.* **2022**, *17* (2), 29–48.

5. For formulations **with existing synthesis parameters at all length scales and for formulations with missing synthesis parameters at some length scales**, should provide new synthesis parameters to design cell with desired property **(Generative modelling, Bottom Up approach, Dataset augmentation, Completing the Dataset)**

6. Take in high level design parameters (Ex: form factor, choice of HTM/ETM, perovskite formulation etc) and predict latent variables at each scale **(Top Down approach)**

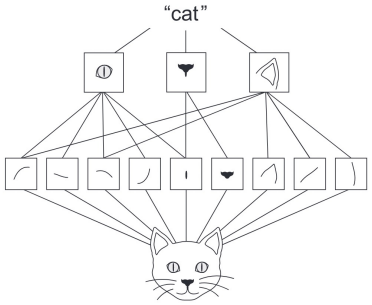
7. Provide a measure of how much information from previous AE's is contained in latent variables of current AE.





Convolutional layers

- Images in the real world are translationally invariant. Advantage of conv nets is that once it detects a pattern in one part of the image it can detect it anywhere else in other images. Problem with dense net is that it would have to learn it anew if detected elsewhere in the image. Another advantage of conv nets over dense nets for images is that they can learn spatial hierarchy of features.
- Every node in the convolutional layer/ filter is not connected to every other node in the input image (unlike in dense layers) , instead each node is connected to ones in its "**receptive field**" (see figure below).
- Each of the neuron's weights (for example the blue and red squares in the first figure) is the size of the 3x3 receptive field. These weights are called **filters or convolutional kernels.**
- For example two possible weights could be something like this :



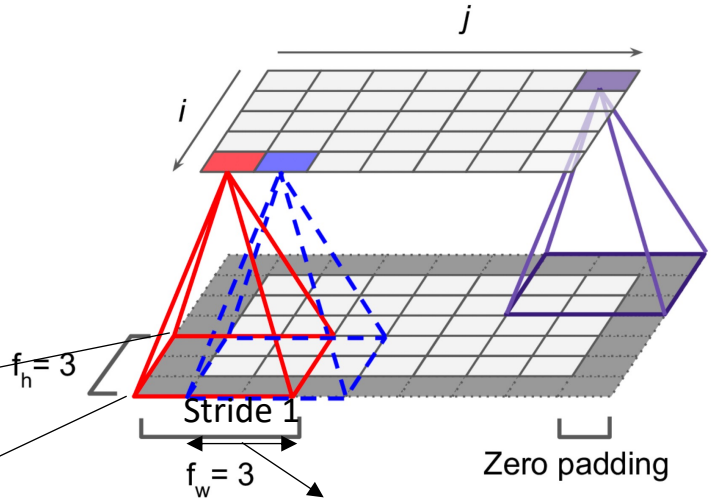
0	1	0
0	1	0
0	1	0

Vertical filter

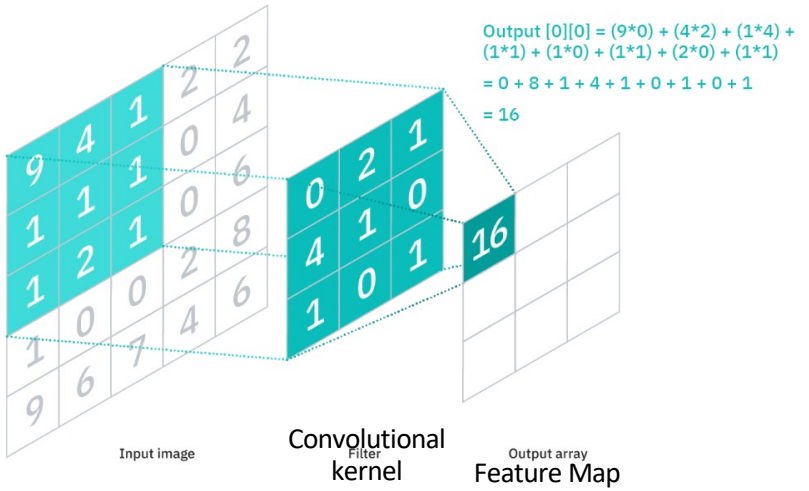
0	0	0
1	1	1
0	0	0

Horizontal filter

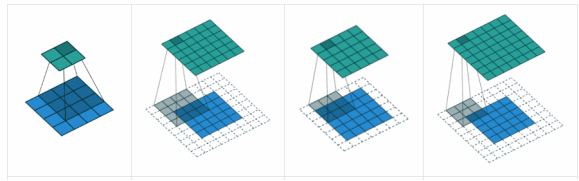
3 x 3 convolutional kernels/filters



This is stride and there can be different stride for height and width individually



- Second figure shows how the convolution takes place. It should be noted that the same filter sweeps across the entire image. Hence all nodes share the same parameters. This is called "**parameter sharing**".
- A layer of neurons all using the same filter output a "**feature map**" which highlights areas in the image that activate the filter the most.
- We do not have to provide the filters and the network automatically learns them by itself by back prop.
- Hyperparameters involved with 2D convolutional layers are
  - Number of filters : 3 distinct filters would yield 3 different feature maps creating a depth of 3. E
  - Stride : Stride is the distance or number of pixels the kernel moves over. Typically 1 or 2. Note there can be different stride values for both the height and width of the image.
  - Padding : Employed when the filter does not fit within the input image. This sets all elements that fall out of input matrix to 0 producing a larger or equally sized input. Should not be a problem if you resize and choose a kernel size that is divisible by the the kernel size.



Num of feature extractors you want (introduces depth) (tensor becomes 4D if you include minn batch)

Determine output size

Conv2D(num filters, kernel size, stride, padding)

- Different padding types supported by tensorflow are valid (no padding) and same (padding same no. of zeros to top/bottom and left/right)
- Just like stacking multiple feature maps on top of one another we can stack multiple convolutional layers on top of one another where the lower level feature maps extract low level features which are then input to downstream feature maps to extract higher level features.
- Generally you would go on increasing the number of filters as you get deeper into the network as the number of low level features is small but there are many ways you can combine them to create high level features. Generally the number of filters are doubled after each 2x2 pooling layer since pooling layer divides each spatial dimension by 1/2.

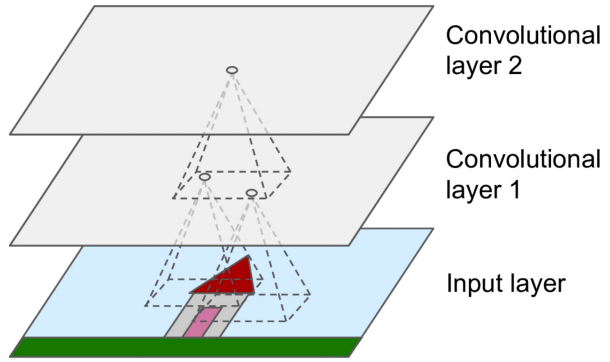
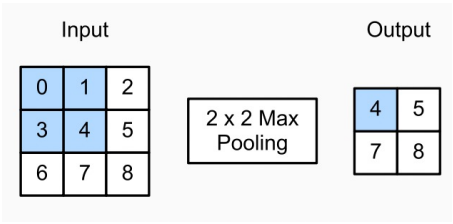
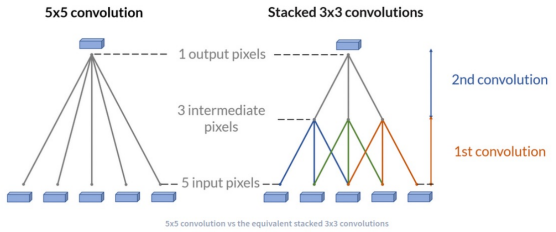


Figure 14-2. CNN layers with rectangular local receptive fields

Pooling layers

- Each neuron in the pooling layer is connected to a few neurons in the previous layer just like in a convolutional layer.
- However, the main idea is to subsample the image to decrease the computational load.
- Hyperparameters involved with Pooling layers are :
  - Padding, Stride and size of pool kernel.
- There are no weights for neurons in the pooling layer. There are two variants in pooling. Max and Mean Pooling. In Max Pooling only the max value makes it to the next layer (most common)





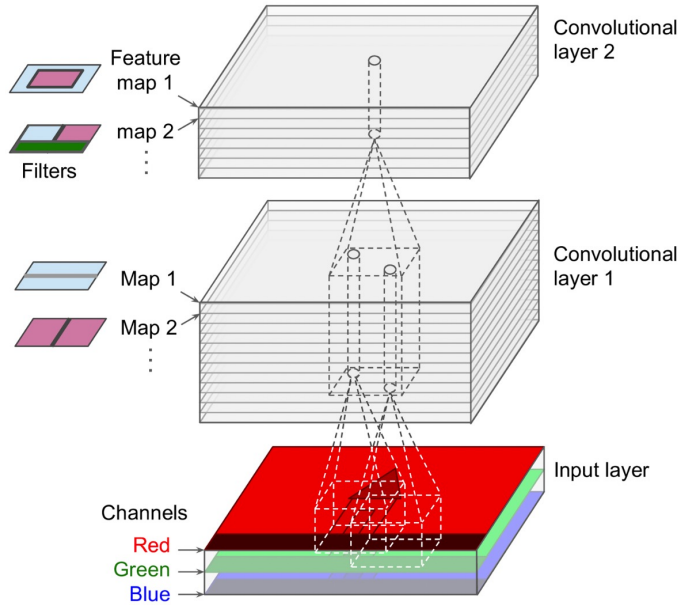


Figure 14-6. Convolutional layers with multiple feature maps, and images with three color channels

Flatten layers

- The last layer in Conv Net.
- The pooling and convolutional layers are 3D layers (height, width, filters). We flatten these 3D layers to get a 1D layer form which we can extract the latent space and attributes of interest.

```
3D Numpy array
[[[ 0  1]
  [ 2  3]
  [ 4  5]]

 [[ 6  7]
  [ 8  9]
  [10 11]]]
```

→ `[ 0  1  2  3  4  5  6  7  8  9 10 11]`

Tips and tricks when building convolutional nets

- Use odd sized kernel for example 3x3, 5x5, 7x7 as these ensure that the output pixel is symmetric about the previous layer pixels.
- A single 5x5 convolution layer can be replaced with stacked 3x3 convolution layers.

Formula to predict output dim of conv2D layer (for valid padding and sq images)

$$\text{Out} = \frac{\text{In} - \text{kernel}}{\text{Stride}} + 1$$

Formula to predict output dim of conv2DTranpose layer (for valid padding and sq img)

Out = Stride x (In – 1) + kernel

Typical CNN architecture

- General workflow is you start with 2D conv layers with certain number of filters say 64 and with a large kernel size (5x5, 7x7) with ReLu activation function.
- The output of this Conv 2D layer can be passed to another Conv2D layer with a larger number of filters or to a max pool layer to reduce the dimensionality of the image. It should be noted if passing to max pool the number of filters will remain the same and only the height and width decrease by half.
- Example CNN architecture LeNet-5.

Layer	Type	Maps	Size	Kernel size	Stride	Activation
Out	Fully connected	–	10	–	–	RBF
F6	Fully connected	–	84	–	–	tanh
C5	Convolution	120	1 × 1	5 × 5	1	tanh
S4	Avg pooling	16	5 × 5	2 × 2	2	tanh
C3	Convolution	16	10 × 10	5 × 5	1	tanh
S2	Avg pooling	6	14 × 14	2 × 2	2	tanh
C1	Convolution	6	28 × 28	5 × 5	1	tanh
In	Input	1	32 × 32	–	–	–

Pooling and 2D conv layers have same number of filters

Convolutional Autoencoders (Down sampling)

- Autoencoders need to downsample. There are two ways to downsample using strided convolutions or use max pooling.
- With Max/ Average pooling there are no parameters being learned. They simply take the max or average value of the values present in the kernel.
- With strided convolution however weights are being learned and the image is getting down sampled simultaneously because of the stride.
- However its more informative to look at maximal feature presence via MaxPool layers than using strided convolutions or average pooling which may cause you to miss certain features or dilute them.

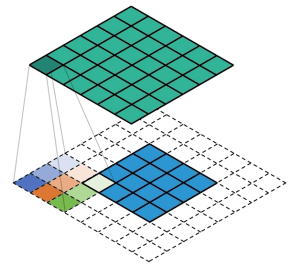
Convolution : $Y = K*X \equiv Y' = WX'$

X' is the flattened representation of input (4x4 -> 16x1)
W is the matrix representation of kernel (4x16);
Y' is the flattened representation of output Y (4x16 . 16x1 -> 4x1 -> 2x2)

Transposed Convolution : $Z = K*Y \equiv Z' = W^TY'$

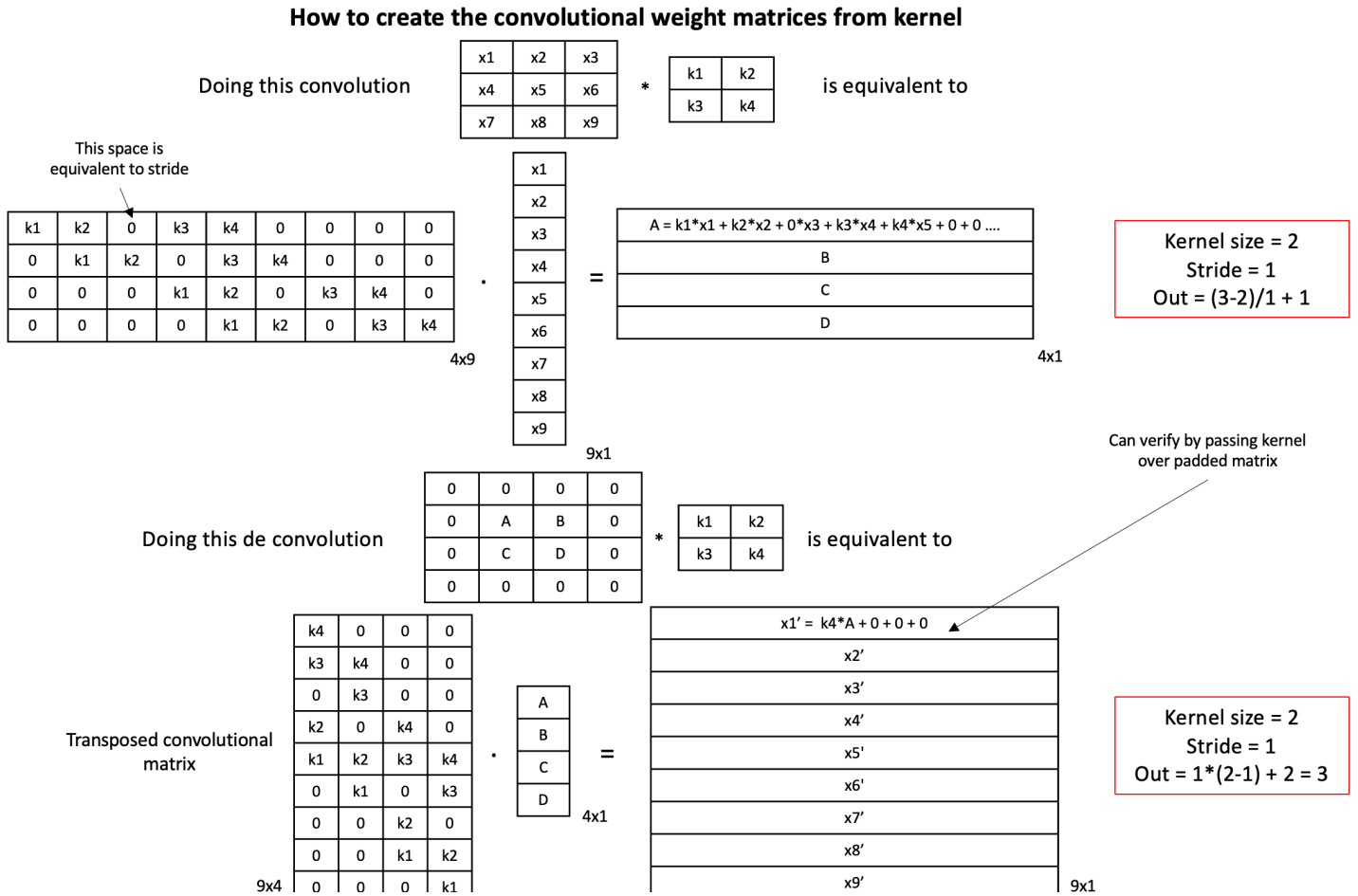
Y' is the flattened representation of input (2x2 -> 4x1)
W^T has the transposed dimensions as that of convolutional kernel (16x4)
Z' is the flattened representation of output (16x4 . 4x1 -> 16x1 -> 4x4)

conv2DTranspse is equivalent to conv2D with padding applied on all edges

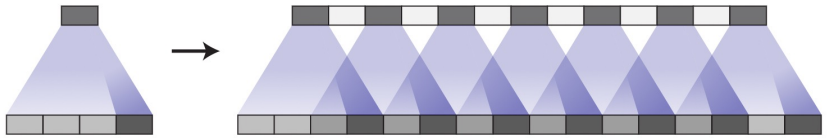


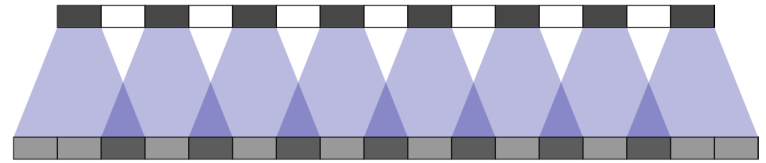
Convolutional Autoencoders (Up sampling)

- While conv2D takes care of extracting important patterns from the 2D image and creating a latent representation. The next step is then how do we take the latent representation and upsample to original image.
- There are many methods for performing up sampling such as using i) nearest neighbours ii) Bi linear interpolation iii) Bi cubic interpolation.
- However these methods contain no learnable parameters. Conv2DTranspose on the other hand has learnable parameters in the convolutional kernel/ filter and has the same parameters as that of Conv2D (i.e (num filters, kernel size, stride, padding={valid, same}) which define size of output layer. However we will see that using deconv lead to formation of checkboard artifacts in images.
- **Conv2DLayer : many-to-one ; Conv2DTranspose : one-to-many**
- To understand how conv2DTranspose works we need to revisit convolutional kernel and view the convolution operation as actually a matrix multiplication operation. See note beside. **Usually Z is not equal to X and hence we cannot call transposed convolution as deconvolution**.

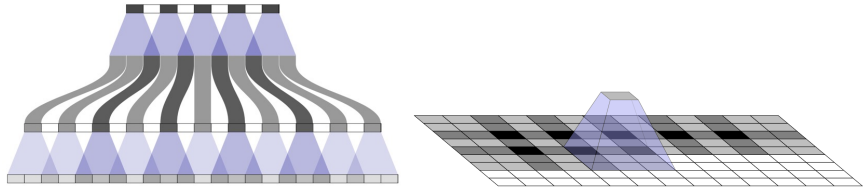


- However, deconvolution suffers from creating checkerboard artifacts in the images. This occurs due to uneven overlap when kernel moves from one pixel to the next in the low dimensional input. It happens when the kernel size is not divisible by the stride. (For ex: kernel of size 3 is not divisible by stride 2) [1]
- Since the kernels contain weights that can be learned. The model can learn to cancel out some of these checkerboard artifact effects but cannot get rid of them completely. Moreover, typical architectures contain multiple such deconvolution layers and hence these checkerboard artifacts get compounded creating more complex artifact structures.



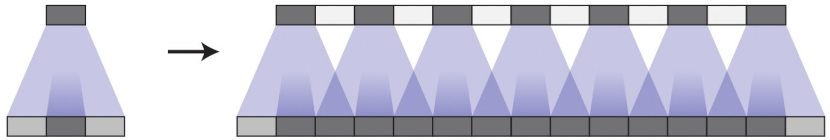


1D Conv Kernel size 3 ; Stride 2

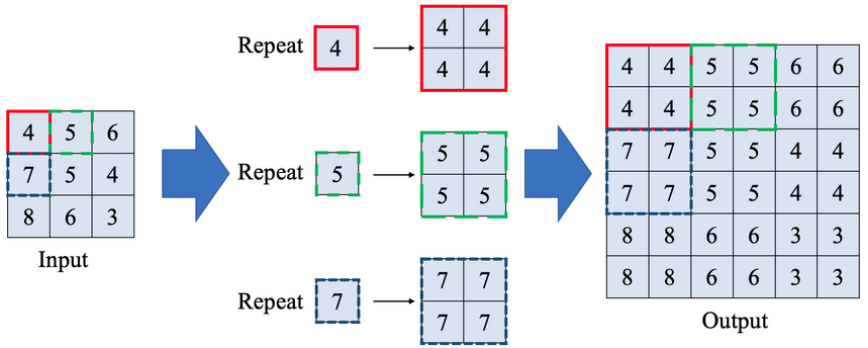


Multiple layers of 1D deconvolution stacked on top of one another

- If you consider even kernels which have sizes that are divisible by the stride you can still get uneven overlap behaviour.



- One way to get rid of these checkboard artifacts is to do resize-convolution instead of deconvolution. Resizing can be done using the resize layer in TF or using the Upsampling2D layer. The Upsampling2D layer can be thought of as inverse of MaxPooling layer. Both Upsampling2D and resizing use interpolation methods to upsample the data. Upsampling2D however can only take an integer kernel size (2x,3x) while resize takes target height and width.
- Default interpolation in Resize is Bilinear and in Upsampling2D is Nearest Neighbours.



[1] https://distill.pub/2016/deconv-checkerboard/

Back Propagation in Convolutional Layers

Forward Propagation

In — Input dim = 28×28

Conv 1 (C1) — $[k^1_{1p}(5\times5)\ ;\ b^1_p(1\times1)\ ;\ \text{Stride}=1]$ — 24×24×6

Pool 1 (S1) — No weights — 12×12×6

Conv 2 (C2) — $[k^2_{pq}(5\times5)\ ;\ b^2_q(1\times1)\ ;\ \text{Stride}=1]$ — 8×8×12

Pool 2 (S2) — No weights — 4×4×12

Flatten + Concatenate — W(10×192) — 192

Out(ŷ) — 10

Conv 1 (C1)

$$C^1_p = \sigma\left(\sum_{u=-2}^{2}\sum_{v=-2}^{2} I(i-u,j-v) * k^1_{1p}(u,v) + b^1_p\right)$$

(convolution operation)

Analogy: $A^1_i = \sigma(x_i w^1_{i1} + b^1_i)$ $A^1_2 = \sigma(x_1\omega^1_{12} + b^1_2)$ .. $A^1_p = \sigma(x\omega^1_p + b^1_p)$

Notation: $k^{p^{\text{th}}\text{ layer}}_{\text{input filters}\ q^{\text{th}}\text{ output filters}}$

Pool 2 (S1)

$$S^1_p = \frac{1}{4}\sum_{u=0}^{1}\sum_{v=0}^{1} C^1_p(i-u, j-v) \quad (i,j = 1,2\ldots12)$$

Conv 2 (C2)

$$C^2_q = \sigma\left(\sum_{p=1}^{6}\sum_{u=-2}^{2}\sum_{v=-2}^{2} S^1_p(i-u,j-v) * k^2_{pq}(u,v) + b^2_q\right)$$

$$C^2_1 = \sigma\left(\left[\sum_{u=-2}^{2}\sum_{v=-2}^{2} S^1_1(i-u,j-v) * k^2_{11}(u,v) + \sum_{u=-2}^{2}\sum_{v=-2}^{2} S^1_2(i-u,j-v) * k^2_{21}(u,v) + \sum_{u=-2}^{2}\sum_{v=-2}^{2} S^1_3(i-u,j-v) * k^2_{31}(u,v) + \sum_{u=-2}^{2}\sum_{v=-2}^{2} S^1_4(i-u,j-v) * k^2_{41}(u,v) + \sum_{u=-2}^{2}\sum_{v=-2}^{2} S^1_5(i-u,j-v) * k^2_{51}(u,v) + \sum_{u=-2}^{2}\sum_{v=-2}^{2} S^1_6(i-u,j-v) * k^2_{61}(u,v) + b^2_1\right]\right)$$

(k^2_{pq} (5×5×6×12))

Analogy: $A^2_i = \sigma(x_1(\omega^2_{11}) + x_2(\omega^2_{21}) + x_3(\omega^2_{31}) + x_4\omega^2_{41} + x_5\omega^2_{51} + x_6\omega^2_{61} + b^2_1)$

Pool 2 (S2):

$$S^2_q = \frac{1}{4}\sum_{u=0}^{1}\sum_{v=0}^{1} C^2_1(2i-u, 2j-v)$$

Flatten + Concatenate:

In = 4×4×12 Out = 192×1

Output:

In = 192×1 Out = 10×1

Loss: Mean Squared Error

$$MSE = \frac{1}{2}\sum_{i=1}^{10}[\hat{y}(i) - y(i)]^2$$

↳ Batch size = 2 (Take avg of cost over the no. of training examples to calc gradient)

If no. of output neurons = 5 then cost function becomes $\frac{1}{2}\sum_{i=1}^{5}[\hat{y}(i) - y(i)]^2$

Back Propagation

Error in weight $\omega(i,j)$ = $\Delta\omega(i,j)$

$$L = \frac{1}{2}\sum_{i=1}^{10}(\hat{y}(i) - y(i))^2 = \frac{1}{2}\sum\left[\sigma(w(i,j)\cdot f(j) + b(i)) - y(i)\right]^2$$

$$= \frac{1}{2}\left[\sigma(w(1,j)\cdot f(j) + b(1)) - y(1) + \sigma(w(2,j)\cdot f(j) + b(2)) - y(2) + \ldots \sigma(w(10,j)\cdot f(j) + b(10)) - y(10)\right]^2$$

$$\frac{\partial L}{\partial \hat{y}(i)} = [\hat{y}(i) - y(i)]_{10\times1} \qquad \sigma(x) = \frac{1}{1+e^{-x}} \quad \frac{\partial\sigma(x)}{\partial x} = \frac{e^{-x}}{(1+e^{-x})^2} = \sigma(x)[1-\sigma(x)] = \hat{y}(i)[1-\hat{y}(i)]$$

$$\Delta\omega(i,j) = \frac{\partial L}{\partial \hat{y}(i)} \times \frac{\partial \hat{y}(i)}{\partial w(i,j)} \times [\hat{y}(i) - y(i)]\,\hat{y}(i)[1-\hat{y}(i)]\,f(j) = \underbrace{\Delta\hat{y}(i)}_{10\times1}\cdot\underbrace{f(j)}_{192\times1} = \Delta\hat{y}(i)_{10\times1}\cdot f^T(j)_{1\times192}$$

Out → 1920 vector error propagation ∴ $\Delta\omega(i,j)_{10\times192} = \Delta\hat{y}(i)f^T(j)$

Step2: Calculating weight updates for k^2_{pq} (Δk^2_{pq})

$$\Delta k^2_{p,q} = \frac{\partial L}{\partial k^2_{p,q}} = \frac{\partial L}{\partial \hat{y}(i)}\cdot\frac{\partial \hat{y}(i)}{\partial f(j)}\cdot\frac{\partial f(j)}{\partial C^2_q(ij)}\cdot\frac{\partial C^2_q(ij)}{\partial k^2_{p,q}}$$

$$\frac{\partial \hat{y}(i)}{\partial w(i,j)} = \hat{y}(i)[1-\hat{y}(i)]\cdot f(j) \qquad \frac{\partial \hat{y}(i)}{\partial f(j)} = \hat{y}(i)[1-\hat{y}(i)]\cdot w(i,j) \quad \therefore \frac{\partial L}{\partial f(j)} = \frac{\partial L}{\partial \hat{y}(i)}\cdot\frac{\partial \hat{y}(i)}{\partial f(j)} = \Delta\hat{y}(i)_{10\times1}\cdot\omega^T(i,j)_{192\times10}$$

$$\therefore \underbrace{\Delta f}_{192\times1} = \underbrace{\Delta\hat{y}(i)}_{10\times1}\cdot\underbrace{\omega^T(i,j)}_{192\times10}$$

- Reshape error to match Pool shape:

$$\underbrace{\Delta S^2_q}_{4\times4\times12} = F^{-1}(\Delta f)$$

Size = 4×4 (12 4×4 error maps)

- Upsample the error:

$$\Delta C^2_q(i,j) = \frac{1}{4}\Delta S^2_q(\lceil i/2\rceil, \lceil j/2\rceil) \quad i,j = 1,2\ldots8$$

Size = 8×8 (12 8×8 error maps)

*Split the error equally among the 4 cells. In max pooling, pass back error only to max value all others get 0.

Ex: Max pool — filter size = 2, stride = 2

Max Pool Backward Pass — Have to keep track of loc. of max value

$$\Delta k^2_{p,q} = \sum_{i=1}^{8}\sum_{j=1}^{8}\frac{\partial L}{\partial C^2_q(i,j)}\cdot\frac{\partial C^2_q(i,j)}{\partial k^2_{p,q}} \qquad \Delta C^2_q(i,j) = \text{upsample}[F^{-1}(w^T(i,j)\Delta\hat{y}(i))] \text{ (Gradient Tape)}$$

$$= \sum_{i=1}^{8}\sum_{j=1}^{8}\Delta C^2_q(i,j)\cdot\frac{\partial}{\partial k^2_{p,q}}\left[\sigma\left(\sum_{p=1}^{6}\sum_{u=-2}^{2}\sum_{v=-2}^{2} S^1_p(i-u,j-v) * k^2_{pq}(u,v) + b^2_q\right)\right]$$

Why summation?

($k^2_{1,1}(-2,-2)$ in $k^2_{1,1}$ (5×5) → $C^2_q(i,j)$ (8×8) = $C^2_q(\text{out})$)

$C^2_1(1,1) = S^1_1(3,3)k^2_{1,1}(-2,-2) + S^1_1(3,2)k^2_{1,1}(-2,-1) + S^1_1(3,1)k^2_{1,1}(-2,0) + S^1_1(3,0)k^2_{1,1}(-2,1) + S^1_1(3-1)k^2_{1,1}(-2,2) \ldots S^1_1(-1,-1)k^2_{1,1}(2,2)$ (25 terms)

$= \sum_{u=-2}^{2}\sum_{v=-2}^{2} S^1_1(i-u,j-v) * k^2_{1,1}(u,v)$

Now add the bias

$= \left(\sum_{u=-2}^{2}\sum_{v=-2}^{2} S^1_1(i-u,j-v) * k^2_{1,1}(u,v)\right) + b^2_1$

Add non linearity by passing through activation

$= \sigma\left[\left(\sum_{u=-2}^{2}\sum_{v=-2}^{2} S^1_1(i-u,j-v) * k^2_{1,1}(u,v)\right) + b^2_1\right]$

- Take weight $k^2_{1,1}(-2,-2)$ in $k^2_{1,1}$
- To backpropagate error to $k^2_{1,1}(-2,-2)$ we need to know what all pixels it effects in output $C^2_q(\text{out})$
- This is because the error signals follow the same connectivity as the input signals.
- The answer is every pixel in C^2_qout is affected by weight $k^2_{1,1}(-2,-2)$. This is because each output pixel in $C^2_q(\text{out})$ is formed by convolution of $k^2_{1,1}$ with part of input. Hence there is a contribution of $k^2_{1,1}(-2,-2)$ in each pixel in $C^2_q(\text{out})$.

$$C^2_1(1,1) = \sigma\left(\left[\sum_{p=1}^{1}\sum_{u=-2}^{2}\sum_{v=-2}^{2} S^1_1(1-u,1-v) * k^2_{1,1}(u,v)\right] + b^2_1\right)$$

Take filter 1 of 12 — Take value at 1,1 in filter 1 of 12

$$\frac{\partial C^2_1(1,1)}{\partial k^2_{1,1}(-2,-2)} = C^2_1(1,1)[1-C^2_1(1,1)]\cdot S^1_1(3,3)$$

$$\frac{\partial C^2_1(i,j)}{\partial k^2_{1,1}(u,v)} = C^2_1(i,j)[1-C^2_1(i,j)]\cdot S^1_1(i-u,j-v)$$

$$\Delta k^2_{1,1} = \sum_{i=0}^{8}\sum_{j=0}^{8}\underbrace{\Delta C^2_{1,1}(i,j)\cdot C^2_1(i,j)\cdot[1-C^2_1(i,j)]}_{\Delta C^2_{1,\sigma}(i,j)}\cdot S^1_1(i-u,j-v)$$

$$\Delta k^2_{1,1} = \underbrace{S^1_1(i-u,j-v)}_{12\times12} * \underbrace{\Delta C^2_{1,\sigma}(i,j)}_{8\times8} \text{ (Convolution operation!)}$$

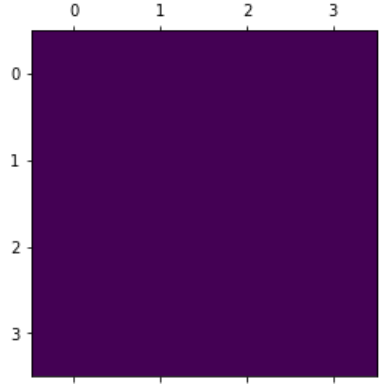
More info at NPTEL-NOC IITM : Backprop in CNNs

Dhruv Batra Back Prop CNN notes

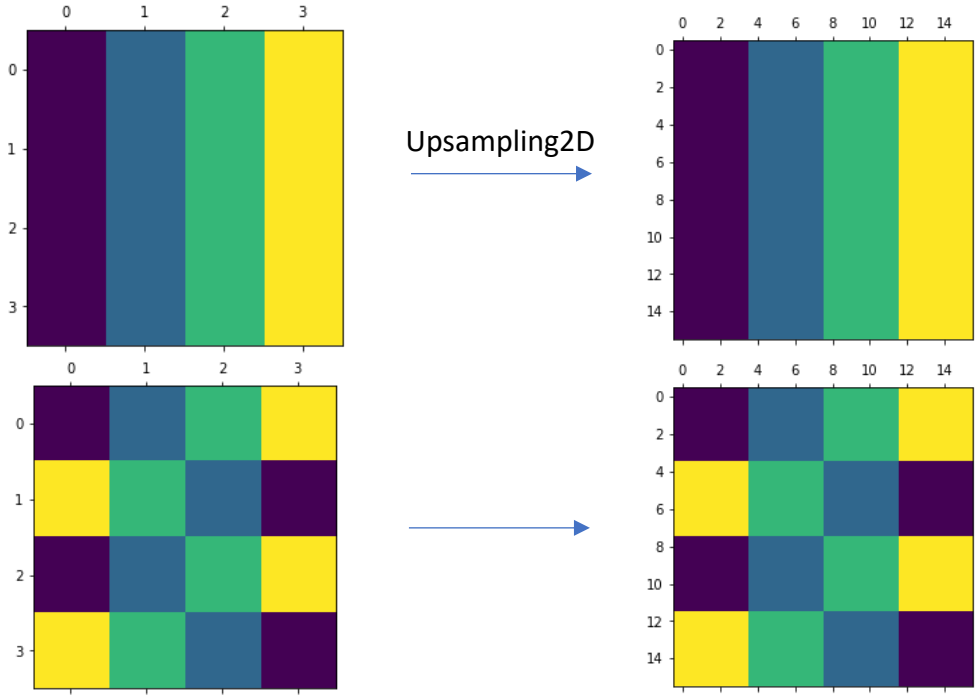
Different interpolation techniques

Nearest neighbour

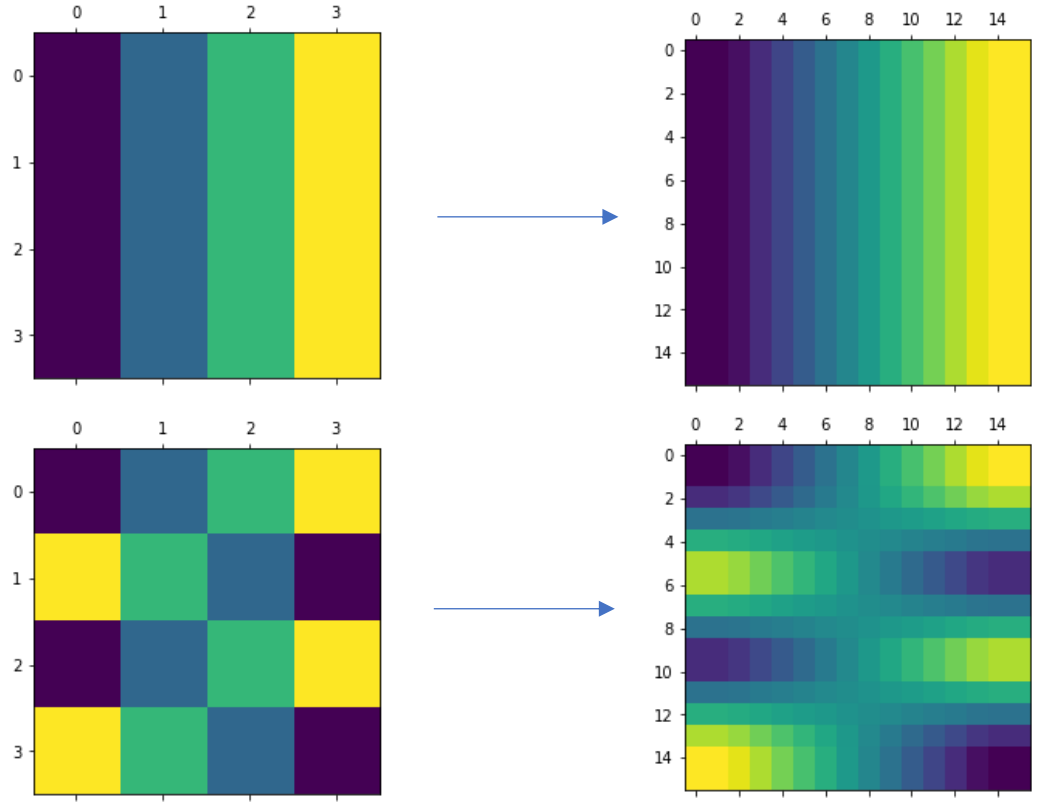
- If input is 1D fro example [[[[1]]]] (1,1,1,1) shape then upsampling to say (1,4,4,1) shape would give something like this



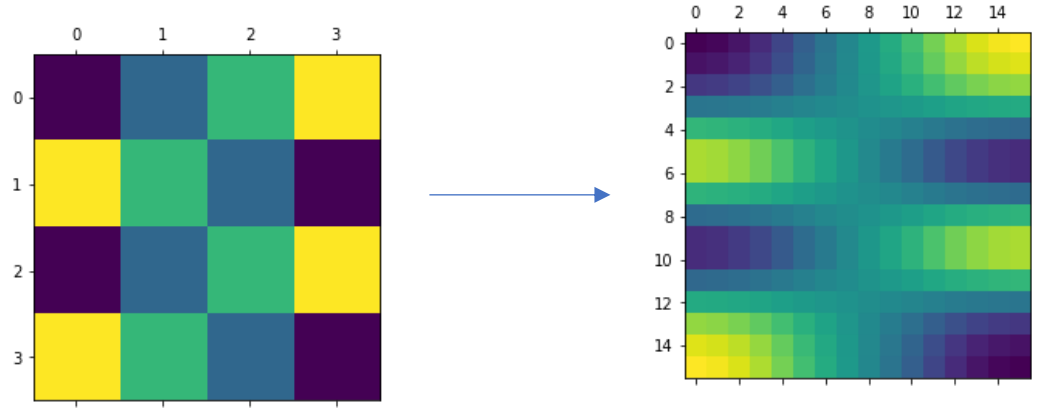
Maybe try using bilinear interpolation and outputting the feature maps learned.



Bilinear

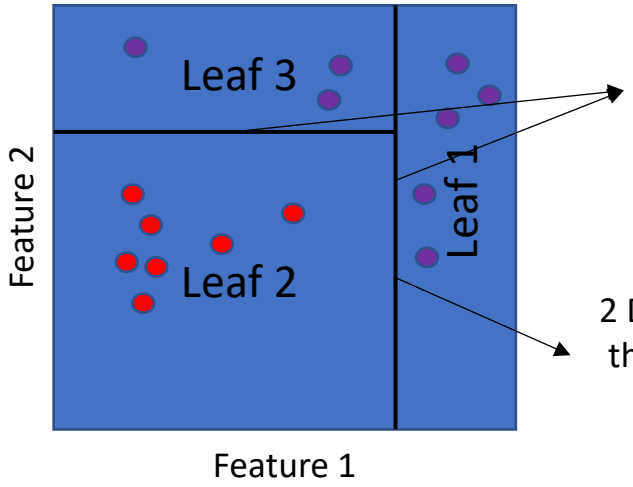


Bicubic



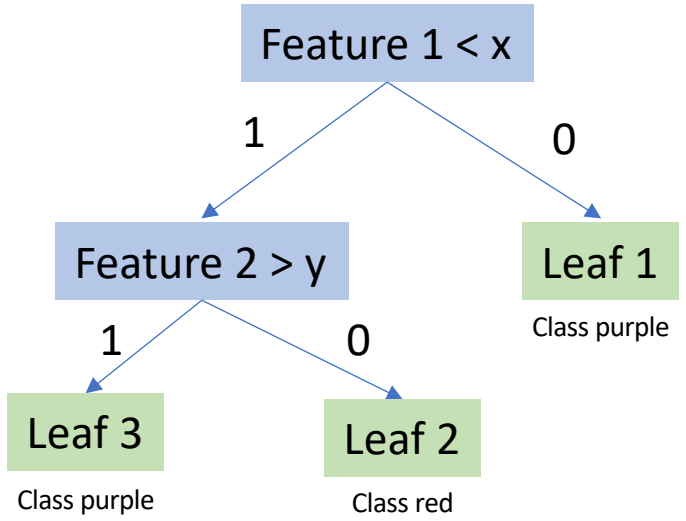
Decision Trees

- Decision Tree is a connected graph (i.e it contains no cycles). Starting node called as root and terminal nodes called as leaves.



Linear Decision Boundaries

2 D feature space then the partioning can be viewed as a line

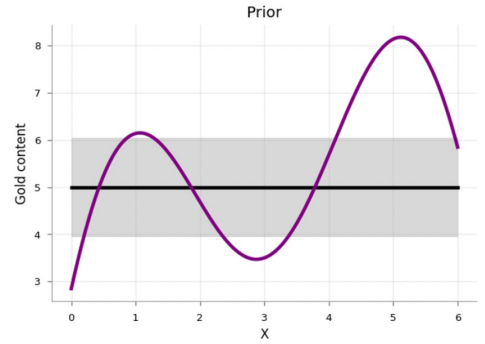


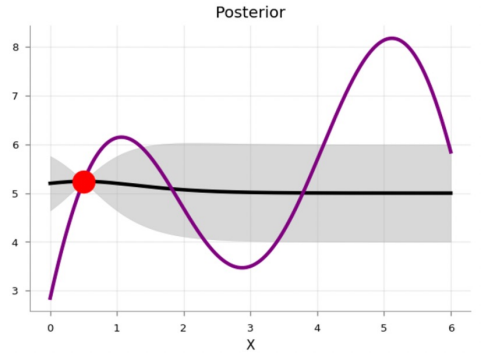
- Number of leaves represent number of decision regions.
- Blue boxes represent the nodes.
  - Contain the split rule
  - Contain number of observations per class.
- Green boxes are the leaves which are the terminal nodes.

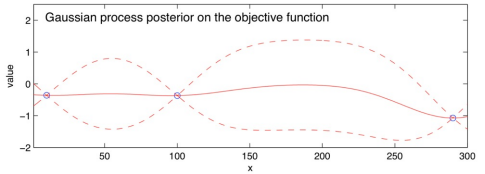
- Splitting Is done to produce a purest possible child node. In the above diagram, x should be chosen in such a way that we can segregate the maximum number of red and purple classes.
- **Growing the tree :** For each node we would like to measure its impurity. In order to split a parent node into child nodes we need to find the average impurity of the child nodes measured as $\text{weighted impurity} = \frac{(\text{num of obs in left}) * (\text{Imurity of left}) + (\text{num of obs in right}) * (\text{Impurity in right})}{\text{num of obs in left+right}}$
- Impurity is measured by Gini index, Entropy etc. Tree is grown by using the greedy algorithm.
- **Controlling the growth :** Tree stops growing if you cannot split a node further. We can prevent the nodes from splitting further by specifying a stopping condition such as
  - Entropy in the node is 0.
  - Contains minimum number of observations.
  - Maximum depth
- **Maximal Tree :** When the number of parameters of the model are more than the number of training samples then we risk at overfitting. Such a decision tree is called a maximal tree. In the case of decision trees these could be the number of nodes.
- **Decision Trees vs MLPs** : Decision Trees are more interpretable and can operate with less number of training points. On the other hand MLPs only the input and output layers are meaningful but the hidden layers not so much and they also require lost of data to learn because of the back propagation algorithm.
- **Other types of decision trees :** Here we looked at decision trees for classification purposes but they can also be used for regression tasks. Look at CART. (Other models are 1D3, C4.5, CART, CHAID, MARS)
- **Decision Trees are quite unstable.** Small variations in the training set can produce different trees and different predictions. Ensemble models such as Boosting and Random forests combine several weak learners to make a prediction.

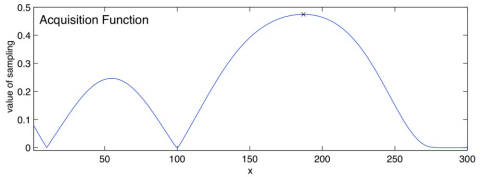
Bayesian Optimization 101

- At a high level the main idea behind Bayesian Optimization is to maximize some expensive to evaluate black box function which I will call as ‘f’ which we typically do not have access to. This is represented by the purple curve in the figure on the right. Unlike in traditional deep learning models where we try to find these functions using the gradient descent algorithm, BO gives you the ability to evaluate for this function and place confidence intervals on the function all while without having to resort to derivative based optimization schemes.
- **Surrogate model :**
  - In order to optimize for this black box function ‘f’ we need to extrapolate our belief about what ‘f’ looks like at points we have not yet evaluated. **This is the job of the surrogate model which quantifies the uncertainty of its predictions in the form of a posterior distribution over function values f(x) for each x**.
  - Often it’s the standard dev. of the posterior distribution which acts as a measure of uncertainty.
  - To start off we need a prior distribution which is the uniform distribution shown in the top figure. After adding an evaluated point indicated by the red circle we update our prior distribution to get the posterior distribution over f(x) indicated in the below figure. (Black line is the mean and the shaded region represents the standard deviation). For example : At 2 the posterior distribution says the function can take on any values between 6 and 4. Each time we encounter a new training point we update our posterior which updates the functional value in the neighboring regions.
  - In traditional variational autoencoders the encoder is essentially learning the mean vector and logvariance vector of the posterior distribution that best matches that of our prior distribution which is multivariate gaussian distribution. However in BO, this posterior distribution is defined over the black box function (f(x)) hence giving rise to the confidence intervals. A similar strategy can be applied in traditional VAE’s to determine the confidence intervals for the parameters of the multi variate gaussian distribution.
  - **In cases where the posterior is implicit in the model then you can just sample from that.**
  - **Surrogate model is typically a Gaussian Process**, in which case the in which case posterior distribution on a finite collection of points is a multi variate gaussian distribution or in other words GP produces a posterior probability distribution on each f(x). GP is specified by a mean function $\mu(x)$ and a covariance kernel k(x,x’) from which a mean vector and covariance matrix Σ is computed for any set of points (x1 to xk)
- **Acquisition function :**
  - This is the second part of Bayesian Optimization which selects test points to evaluate next based on a balance between exploration and exploitation. Some common analytical acquisition functions are Expected Improvement (EI), Upper Confidence Bound (UCB) and Probability of Improvement (PI) all of which require a Gaussian posterior.









Ref :
[1] BoTorch documentation
[2] A tutorial on Bayesian Optimiztaion by Peter I. Frazier
[3] https://distill.pub/2020/bayesian-optimization/
[4]https://www.youtube.com/watch?v=EnXxO3BAgYk

(BayesOpt Continued) Method :

Step 1 : Define the prior distribution by choosing a mean function and covariance kernel

- Consider you have a set of 'n' training points of 'd' dimensions x1,x2, x3,x4 .. Xn ∈ $\mathbb{R}^d$
- We construct a mean vector at each 'x' denoted by μ_o and a covariance kernel Σ_o which relates how close two points are in space based on their functional values.
- Kernel should have the property that is is positive semi definite regardless of collection of points chosen.
- The Resulting prior distribution will look something as below :

$$[f(x_1), f(x_2), f(x_3) \dots f(x_k)] \sim Normal(\mu_o(x_{1:k}), \Sigma_o(x_{1:k}, x_{1:k}))$$

$x_{1:k}$ is sequence $x_1, x_2, x_3 \dots x_k$

Step 2 : Determine the hyperparameters for the kernel

- Optimizing the hyperparameters of the kernel which can be determined by maximum likelihood estimation (MLE) or maximum a priori estimate (MAP) (this method assumes the hyperparams themselves where chosen from a prior distribution)
- The optimizer can be restarted repeatedly be specifying n_restarts_optimizer. First run is conducted from initial hyperparameter values of the kernel and subsequent runs are chosen from range of allowable values.

Step 3 : Determine the posterior distribution based on new evaluation point

- Now we wish to evaluate the value of f(x) at some new point. We need to compute the conditional distribution of f(x) given these observations using Baye's rule.

$$f(x)|\, f(x_{1:k}) \sim Normal(\mu_n(x), \sigma_n^2(x))$$

- The conditional distribution at un evaluated points is also a multi variate gaussian distribution with a mean vector and covariance kernel dictated by location of unevaluated points, locations of measured points $x_{1:k}$ and their measured values f($x_{1:k}$).
- Rather than computing posterior means and variances directly its more faster to compute them Cholesky decomposition and solve a system of linear equations.
- The mean function of the posterior keeps track of the means at each point while the covariance kernel keeps track of the covariance between each training point. In evaluating the mean function for every 'x' you provide you the 'y' values over the domain of 'x'.
- While analytic acquisition functions assume the posterior is multi variate gaussian the Monte Carlo based acquisition functions make no such assumption about underlying distribution and require only that it

- It is a generic supervised learning method designed to solve regression and probabilistic classification problems.
- Advantages of Gaussian Process Regressison :
  - Prediction interpolates the observations (at least for regular kernels)
  - Prediction is probabilistic so that one can compute empirical confidence intervals an d decided based on those if one should refit (based on adaptive learning, online learning) in a certain region.
  - Different kernels can be fit to specifiy the covariance function.
- Disadvantages of GPR :
  - They are not sparse, i.e they use the whole samples/features to perform the prediction.
  - They lose efficiency in high dimensional spaces- namely when the number of features exceeds a few dozen.
- Method (as in scikit learn):
  - The prior mean is assumed to be constant and zero and the prior covariance is specified by passing a kernel object.
  - Hyperparams of kernel are optimized by maximizing the log marginal likelihood (LML) based on the passed optimizer.
  - As the LML can have multiple local optima, the optimizer can be restarted repeatedly be specifyin n_restarts_optimizer. First run is conducted from initial hyperparameter values of the kernel and subsequent runs are chosen from range of allowable values.

Uncertainty prediction

$$y = \hat{f}(l_1, l_2, l_3, l_4, \ldots l_m) + \sigma_{noise}$$

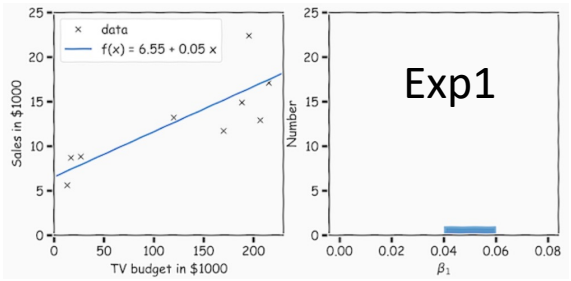
epistemic uncertainity

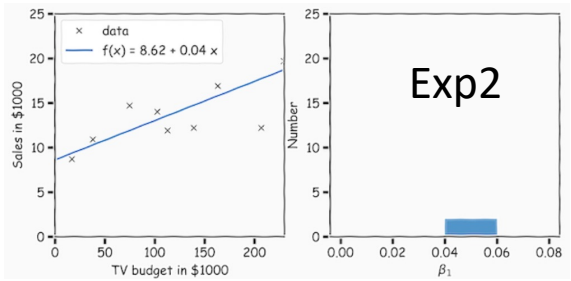
aleatory uncertainity

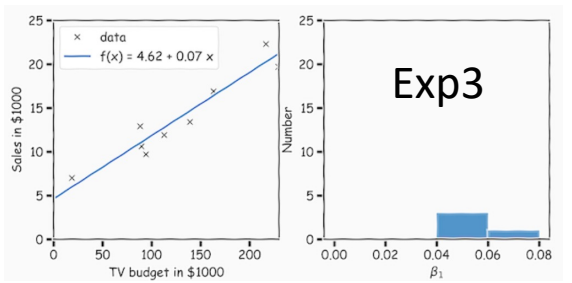
$$\sigma^2 = \sigma_{es}^2 + \sigma_{al}^2$$

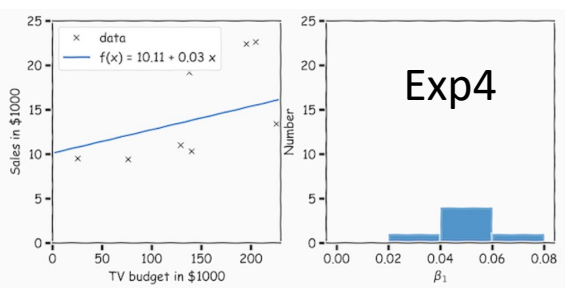
If the noise are independent then the total uncertainty can be calculated as follows

- We do not know the true functional form and the neural networks can only obtain an approximation of the function. This is called epistemic error.
- Also each measurement is associated with some random noise from the environment or from the instrument. This is the observational noise. Also called aleatoric error.
- If we have only one set of observations {X,y} our estimates for the model parameters is only for this set of observables.
- One way to solve this is to use bootstrap to sample multiple different datasets to train the model.





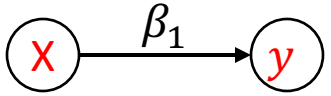




- Problem with this approach is 1) computationally intensive 2) uses maximum likelihood point estimates
- Another approach we can try is to directly calculate the P(beta|data) = $\frac{P(data|beta)*P(beta)}{P(data)}$. The major issue with this approach is what is P(data). This is extremely hard to predict. Difficult to predict the posterior as a closed form. Hence use inference techniques to arrive at an approximation of the posterior.
- Multiple methods exist for uncertainty prediction. Some of them are listed below :
  - Guassian Processes (GP) used in Bayesian Optimization,
  - Bayesian Neural networks
  - Ensemble methods : Taking an average across the predictions from multiple models
  - Constructing Prediction Intervals

[1] Lai, Y.; Shi, Y.; Han, Y.; Shao, Y.; Qi, M.; Li, B. Exploring Uncertainty in Deep Learning for Construction of Prediction Intervals. arXiv April 26, 2021. http://arxiv.org/abs/2104.12953

- **Bayesian Neural Networks** (Models uncertainity based on weights of the NN) **:**
  - The weights of a neural network are the parameters of the function that the model learns to optimize during training so that it can get the best possible prediction that matches close to the true value.
  - Bayesian neural networks introduces uncertainty into those parameters by introducing a prior distribution on network parameters. The model learns a posterior distribution on the weights and then randomly samples from the posterior distribution.
  - Simple example is for a linear regression model where you have a no hidden layer NN.



$$y = \beta_1 X + \beta_0$$

  - In gradient descent approach we learn the optimal values of weights via gradient descent. In Bayesian Neural nets we instead approximate a posterior distribution P(beta|data) and then sample from the distribution using techniques such as MCMC.
  - Main disadvantage is that as the number of parameters grow harder it is to compute posterior.
- **Ensemble models** (Models uncertainty based on model output predictions) **:**
  - Use an ensemble of multiple models trained using the bootstrapping approach
  - A mean or weighted sum of predictions from all models is computed and the standard deviation can be calculated as well.
- **Computing Prediction intervals** (Models uncertainity on output prediction by computing prediction interval) :
  - Predicting an upper and lower bound on prediction in comparison traditional neural networks produce only point estimates.

[1] Jospin, L. V.; Buntine, W.; Boussaid, F.; Laga, H.; Bennamoun, M. Hands-on Bayesian Neural Networks -- a Tutorial for Deep Learning Users. *IEEE Comput. Intell. Mag.* **2022**, *17* (2), 29–48. https://doi.org/10.1109/MCI.2022.3155327.

- Agent performs an action ‘a’ from a set of actions (A) in an environment which takes it from one state to the next and returns to the agent some reward.
- Goal of agent is to maximize the cumulative rewards.
- At each time step agent takes an action on the environment based on its policy $\pi(a_t|s_t)$ and receives a reward r_{t+1} and next observation from the environment s_{t+1}
- Dynamic Programming :
  - Requires model of environment
  - Can obtain optimal policies once we have found the optimal value function $v_*(s)$
  - $v_*(s) = \max_{\mathrm{a}} \mathrm{E}[R_{t+1} + \gamma v_*(S_{t+1}) \,|S_t = s, A_t = \mathrm{a}] = \max_{\mathrm{a}} \Sigma p(s', r \,|s, a)[R_{t+1} + \gamma v_*(S_{t+1})]$ (Bellman Optimality Equation) (Value of state given optimal value selections)
  - $q_*(s, a) = \mathrm{E}\left[R_{t+1} + \gamma \max_{\mathrm{a}'} q_*(S_{t+1}, a') \middle| S_t = s, A_t = \mathrm{a}\right]$ (Value of action in given state)
- Q learning :
  - The goal of a Q learning agent is to maximize the total rewards. It assigns value to an action by determining a potential future award that is added to current award for achieving current state which directly influences the action in the current state. This potential reward is a weighted sum of expected values of the rewards of all future states starting from current state. The ‘Q’ in Q-learning stands for the analytic function that is used to determine the potential future reward.
  - Does not require a model of the environment.
  - $Q(S_t, A_t) <- \ Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_{\mathrm{a}} Q(S_{t+1}, a) \ - Q(S_{t+1}, At)]$
  - Must initialize Q(S,A) for all states in S^+ and action ‘a’ in A(s) ; Q(terminal state, a) = 0
  - Choose A based on current state S according to policy derived from Q (Ex: epsilon-greedy)
  - Take A, Observe R and state S’
  - Update Q(S,A) <- Q(S,A) + α[R + $\gamma \max_{\mathrm{a}} Q(S', A) \ - Q(S, A)$]
  - Continue until you reach terminal state.
- DQN agent :
  - Uses Q-learning combined with deep learning.
  - DQN replaces an exact value function with a function approximator.
  - In Q – learning you are updating one action state pair but in DQN you are updating multiple action state pairs at once. This will affect the action values of next state instead of them being stable as in Q-learning.
  - Using a stable target network as your error measure is one way to combat this effect.

- DDPG (Deep Deterministic Policy Gradient) agent :
  - Actor-critic RL algorithm that seeks to find optimal policy to maximize expected cumulative reward.
  - Combines ideas from DQN and Deterministic Policy Gradient (DPG). It can act in continuous action spaces since it uses DPG.
  - Has two deep networks. One is called the actor and the other called the critic.
  - Actor produces an action given the state. Actor loss is computed by mean of value given by Critic network for actions taken by actor network. We want to maximize this quantity.
  - Critic sees if the action is good (positive value) or bad (negative value) based on given the state. Using two target networks adds stability to the training. Critic loss is calculated as MSE(y - Q(s,a)) where Q(s,a) is the action value predicted by the target network. 'y' is a moving target the critic network tries to achieve. We want to update this moving target as slow as possible by slowly updating the target network.
  - Also it uses experience replay. Store a tuple (state, action, reward, next state) and instead of learning from only previous experience learn by sampling all the experience accumulated so far.

Reinforcement Learning in MATLAB

- rlNumericSpec(dimension, name, description)
  - Specifies continuous action or observation data specifications for reinforcement learning environments

- Need to code the parameter set of the optimization problem as a finite length string over a finite alphabet.
- GA starts with a population of coded samples and thereafter generates successive populations of strings. One important point here is the resultant accuracy and efficiency depends on the diversity of the starting population.
- Unlike gradient based optimization schemes where a surrogate function (Ex: mae, mse) is used as the optimization function to guide the fitting of function mapping from input space to prediction space, GA directly uses the function input and outputs and optimizes over which inputs and combination of inputs maximizes or minimizes the function. There is no calculation of any gradients to guide the optimization process.
- New string populations are created by 3 mechanisms :
    - Reproduction : Choose string to go to next generation based on function value. Higher the function value greater the chance it has in surviving to next generation.
    - Crossover : Randomly select location on two strings to break string and switch positions
    - Mutation : Introduce a point mutation located anywhere on the string (Ex : bit flip)
- Example : Maximize f(x) = x2 in interval [0,31]
    - Optimization function : x2
    - Optimization variable : x
    - Code parameter x as a string. We can code each number as a binary string.
    - Perform this coding for every sample.
    - Choose a random starting population (4 samples). One way to do this is flip a coin 20 times.
    - For each of the random starting samples calculate the function value.
    - Now produce the next generation of strings by using the reproduction operator. A simple way to visualize the reproduction operator is to think of a weighted roulette wheel where the weights are dictated by the function value. Spin the roulette wheel 4 times to generate the next generation of 4 ‘5-bit’ samples.
    - Strings to mate are chosen randomly (for ex through coin tosses), crossing sites are chosen again randomly.
    - Mutation is performed on bit-by-bit basis.

Ref : Goldberg, Genetic algorithms in search, optimization and machine learning

Working with imbalanced datasets

Trick 1 : Init well. Initialize your bias well

- Verify loss at initialization :
  - For softmax : initial loss should be $-\log\left(\frac{1}{n_{classes}}\right)$

$$\text{CCE} = -\sum_{i}^{K} P_{true}(G = i|X)\log(P_{pred}(G = i|X))$$ Assumption all bias are same

$$P_{pred}(G = i|X) = \frac{e^{b+WX}}{\sum_i^K e^{b+WX}} \approx \frac{e^b}{\sum_i^K e^b} = \frac{1}{K}\ (K\ is\ number\ of\ classes)$$

During the intial stages of training most of the weights will be close to 0 and we will only the initialized bias. $P_{true} = [0,0,1,0\ \ldots..0]$ vector of length 'K'

$$\text{CCE} = -\sum_{i}^{K} P_{true}(G = i|X)\log\left(\frac{1}{K}\right) = -\log\left(\frac{1}{K}\right)$$

- Initialize weights and biases properly. (See previous page about different weight initialization schemes for different activation functions). If you have an imbalanced number of classes set the bias on your logits accordingly.
- For example in Ravithree's nanoparticle dataset water has 332 sample while others have less than 30 samples per class.

[DMF, ethanol, formamide, glycerol, none, water]

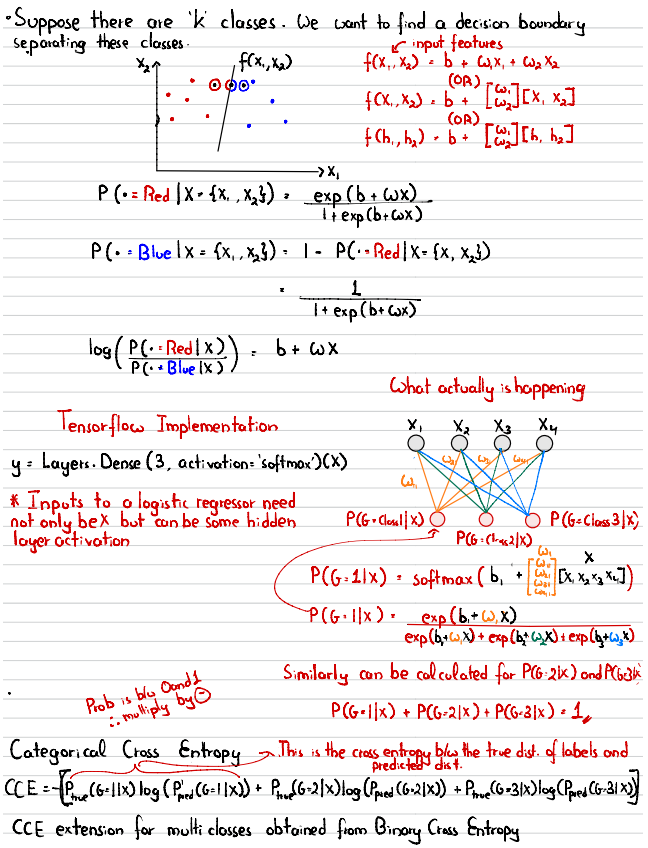
$$P_{water} = \frac{\text{num water samples}}{\text{total samples}} = \frac{332}{410} = 0.81 \approx \frac{e^b}{\sum_i^K e^b}$$

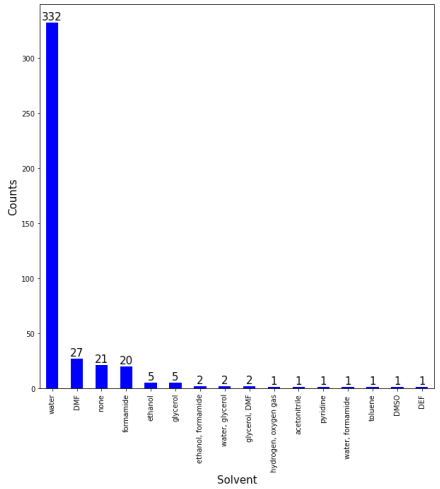
$b_1 = \log(0.81a)$ where $a = \sum_i^K e^b$

Assume $\sum_i^K e^b = 1$

$$\text{b} = \log(\text{P} * \text{a})$$

[DMF, ethanol, formamide, glycerol, none, water] = [-0.21, -2.72, -2.98, -3.02, -4.42, -4.42]





Tensorflow link on imbalanced datasets : https://www.tensorflow.org/tutorials/structured_data/imbalanced_data

Andrej Karpathy blog : http://karpathy.github.io/2019/04/25/recipe/

Trick 2 : Class weighting and Sample weighting

- Class weights :
  - Weight underrepresented classes
  - Useful for categorical samples
  - Passed as a dictionary where the class index (integer) is mapped to a weight value. Used for weighting the loss function only during training.
  - Pass as arguments to model.fit(class_weight:dict)
- Sample weights :
  - Weight samples that lie in a certain range
  - Useful for numerical samples.
  - Passed as numpy array of weights for training samples. Used for weighting the loss only during training.
  - Pass a numpy array with the same length as the input samples to model.fit(sample_weight:list)
- Weighted metrics is a variable passed to compile(). List of metrics to be evaluated and weighted by 'sample_weight' or çlass_weight'during training and testing.
- There are different weighting schemes :
  1. Inverse number of samples : 1/(Number of samples in class C)
  2. Inverse of square root of number of samples
  3. Effective number of samples :
     - Sample_weight = 1/E ; E = (1 – betaˆ(num in class C))/(1 – beta). Different beta values are 0.9, 0.99, 0.999, 0.9999
     - Ref : https://openaccess.thecvf.com/content_CVPR_2019/papers/Cui_Class-Balanced_Loss_Based_on_Effective_Number_of_Samples_CVPR_2019_paper.pdf

Trick 3 : Resampling

- See guide here : Involves using tf.data https://www.tensorflow.org/guide/data#resampling

Trick 4 : Undersample Majority Class and Oversample Minority Class

- Remove certain number of samples associated with the majority class. Oversampling for minority class entails repetition of samples associated with the minority class.
- Oversampling : See SMOTE

Ref : https://wandb.ai/authors/class-imbalance/reports/Simple-Ways-to-Tackle-Class-Imbalance--VmlldzoxODA3NTk
Ref : https://www.ibm.com/support/pages/calculating-fractional-weights-correct-under-and-over-sampling

Handling missing data in a dataset

- **Univariate feature imputation :** Simplest one involves imputing NA values with mean, median or value with most occurrence.
- **Multivariate feature imputation :** Model each feature with missing values as a function of other features. Imputation performed for multiple rounds. Common practice to perform several rounds of imputation rather than one. Each of the different feature column variants obtained after each imputation round can be sent through the pipeline to analyze the effect of imputing the missing values.
- **KNN imputation (Nearest neighbour imputation) :** Impute using the average value of k nearest neighbours

Relation between batch sizes and a learning rates

- There is high correlation between learning rate and batch size. When learning rate is high batch size should be high and when its low batch size should be low.
- Batch sizes typically go in the order of powers of 2 to take advantage of GPU.
- Gradient descent can be done using a single sample (stochastic gradient descent (SGD)), entire training set (batch gradient descent) or mini batch gradient descent which uses mini batches with size varying between 1 and total dataset.
- For small mini batches there will be larger variations between the data and hence the gradient. So while doing the update a smaller learning rate should be used.
- Generally lower learning rates and bathc sizes should be used for training CNN's.

Regularization

- Can be of two types : Implicit and Explicit.
- Implicit Regularization : Model architecture itself can be a regualrizer, model early stopping
- Explicit Regularization : Weight decay, dropout, data augmentation (random resizing, cropping)
- Ref : https://arxiv.org/pdf/1611.03530.pdf

L1 Regularization

$$\Omega(\theta) = ||\omega||_1 = \sum_i |\omega_i|$$

Normalized cost function

$$\tilde{J}(\omega; X, y) = \alpha \sum_i |\omega_i| + J(\omega; X, y)$$

Cost function which contains the reconstruction + classification loss

$$\nabla_w \tilde{J}(\omega; X, y) = \alpha * sign(w) + \nabla_w J(\omega; X, y)$$

- L1 regularization is a parameter norm penalty.
- L1 is essentially a sum of the absolute values of the weight.
- Compared to L2 in L1 the regularization contribution no longer scales linearly with each weight.
- LASSO (Least Absolute Shrinkage and Selection Operator) integrates an L1 penalty with a linear model and least squares function.

L2 Regularization

$$\Omega(\theta) = 0.5||w||_2^2 = \sum_i |\omega_i||\omega_i|$$

Normalized cost function

$$\tilde{J}(\omega; X, y) = \alpha/2 w^T w + J(\omega; X, y)$$

Cost function which contains the reconstruction + classification loss

$$\nabla_w \tilde{J}(\omega; X, y) = \alpha * w + \nabla_w J(\omega; X, y)$$

Weight update is : $w \leftarrow w - \in (\alpha w + + \nabla_w J(\omega; X, y))$

- L2 regularization is a parameter norm penalty.
- L1 is essentially a sum of the squares of the weight values
- Regularization scales linearly wrt each weight.
- Also called as ridge regression or Tikhonov regularization.

The typical normal equation in case of regression loss is $w = (X^T X)^{-1} X^T y$

For l2 loss : $w = (X^T X + \alpha I)^{-1} X^T y$

- Optimization using l2 loss causes the input to be perceived as having higher variance because of the added αI to $X^T X$ (measure of variance in the data).
- This causes it to shrink the weights on features whose covariance with output target is low compared to the added variance.

Sparse Representations

- L1 regularization promotes sparsity in the parameters. However we can also impose sparsity constraints on the activations also which indirectly imposes a complicated penalty on the model parameters.

Sparse kernel matrix

$$\begin{bmatrix} 18 \\ 5 \\ 15 \\ -9 \\ -3 \end{bmatrix} = \begin{bmatrix} 4 & 0 & 0 & -2 & 0 & 0 \\ 0 & 0 & -1 & 0 & 3 & 0 \\ 0 & 5 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & -4 \\ 1 & 0 & 0 & 0 & -5 & 0 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \\ -2 \\ -5 \\ 1 \\ 4 \end{bmatrix} \quad (7.46)$$

$\boldsymbol{y} \in \mathbb{R}^m$ $\quad \boldsymbol{A} \in \mathbb{R}^{m \times n}$ $\quad \boldsymbol{x} \in \mathbb{R}^n$

Sparse activation matrix

$$\begin{bmatrix} -14 \\ 1 \\ 19 \\ 2 \\ 23 \end{bmatrix} = \begin{bmatrix} 3 & -1 & 2 & -5 & 4 & 1 \\ 4 & 2 & -3 & -1 & 1 & 3 \\ -1 & 5 & 4 & 2 & -3 & -2 \\ 3 & 1 & 2 & -3 & 0 & -3 \\ -5 & 4 & -2 & 2 & -5 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 2 \\ 0 \\ 0 \\ -3 \\ 0 \end{bmatrix} \quad (7.47)$$

$\boldsymbol{y} \in \mathbb{R}^m$ $\quad \boldsymbol{B} \in \mathbb{R}^{m \times n}$ $\quad \boldsymbol{h} \in \mathbb{R}^n$

- Norm penalty on representations is obtained by adding to the loss function $\alpha\Omega(h)$
- $\Omega(h) = ||h|| = \sum_i h_i$

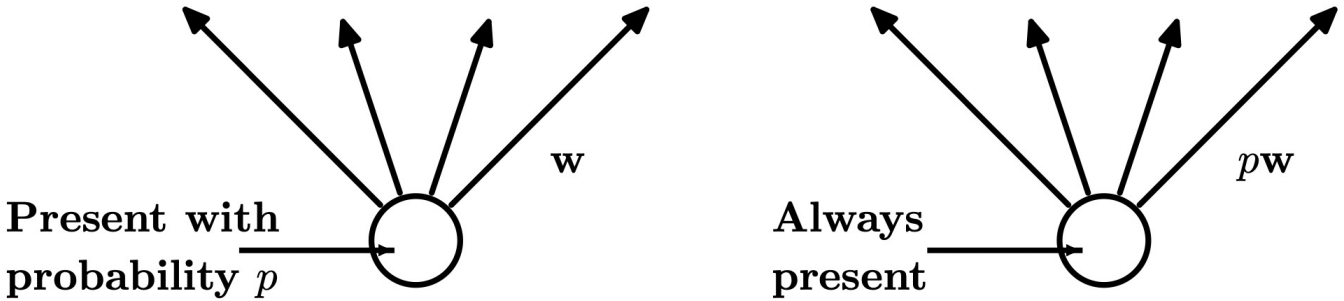
Source : https://www.deeplearningbook.org/contents/regularization.html

Noise Robustness

- Noise injection can be much more powerful than simply shrinking the weights especially when added to hidden units.
- Adding noise to weights :
  - The Bayesian treatment of learning would consider the model weights to be uncertain and representable via a probability distribution that reflects this uncertainty. Adding noise to the weights is a practical, stochastic way to reflect this uncertainty.
  - This form of regularization encourages the parameters to go to regions of parameter space where small perturbations of the weights have a relatively small influence on the output. In other words, it pushes the model into regions where the model is relatively insensitive to small variations in the weights, finding points that are not merely minima, but minima surrounded by flat regions
- Adding noise at the output targets :
  - For small $\in$ we can assume the training label on y is correct with probability 1 - $\in$.
  - Consider a softmax output with ‘k’ values. **Label smoothing** replaces the hard 0/1 output with soft outputs of 1 - $\in$ and $\in$ / (k-1**)**

Dropout

- Dropout only applied during training where feature detectors are deleted with a certain probability say ‘p’ and the remaining weights are trained by backprop.
- Dropout can be applied to weights and also the output activations.
- Dropout performs a kind of geometric averaging across over the ensemble of possible sub networks (see figure on the right)
- **Motivation behind dropout is that it prevent co adaptation among hidden units. Each hidden unit must learn to perform well with a randomly chosen sample of other units.**



(a) At training time (b) At test time

- Applying dropout essentially means sampling a thinned network from the original network. A neural net with n units can be seen as an ensemble of 2^n subnetworks.
- The networks all share weights so total number of parameters is still O(n^2). For each training case a new subnetwork is sampled and trained. So training with dropout can be seen as training an ensemble of 2^n subnetworks with extensive weight sharing. Each thinned network gets trained rarely.
- At test time a single neural net is used and all the units are retained. However the weights connected to that unit are rescaled with probability ‘p’ that the unit was retained during train time. This ensure that the ‘expected’ output is the actual output at test time.

Activity of unit 'i' in layer 'h' is $S_i^h(I) = \sum_{l<h} \sum_j \omega_{ij}^{hl} S_j^l \quad S_j = I$

Dropout applied to units can be written as

$S_i^h = \sum_{l<h} \sum_j \omega_{ij}^{hl} \delta_j^l S_j^l$ δ_j^l is a gating 0-1 Bernoulli variable with $P(\delta_j^l = 1) = p_j^l$

δ_j^l are independent of each other, independent of activations, ind. of weights

Dropout applied to weight connections

$S_i^h = \sum\sum \delta_{ij}^{hl} \omega_{ij}^{hl} S_j^l$

Expectation of activity of all units, taken over all possible realizations of the gating variable (all possible realizations of subnetworks)

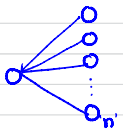
$E(S_i^h) = \sum_{l<h} \sum_j \omega_{ij}^{hl} p_j^l E(S_j^l)$

Can calculate ensemble average by replacing $\omega_{ij}^{hl} \rightarrow \omega_{ij}^{hl} p_j^l$

Consider 1 logistic unit with 'n' inputs

$S = \sum_{i=1}^{n} \omega_i I_i$

$O = \sigma(S) = \frac{1}{1+ce^{-kS}}$



$= 2^n$ possible subnetworks

$m = 2^n$

'm' different outputs $O_1, O_2 \dots O_m$ corresponding to 'm' diff sums $S_1, S_2 \dots S_m$ each with probability $P_1, P_2 \dots P_m$

$E = \sum P_i O_i \quad E' = \sum (1-P_i) O_i$ Weighted Geometric Mean (WGM) $G = \prod_i O_i^{P_i}$

Normalized WGM $= \frac{G}{(G+G')}$

$NWGM(O_1, O_2 \dots O_m) = \sigma(E(S))$

[1] Baldi, P.; Sadowski, P. J. Understanding Dropout. 9.
[2] Srivastava, N.; Hinton, G.; Krizhevsky, A.; Sutskever, I.; Salakhutdinov, R. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. 30.

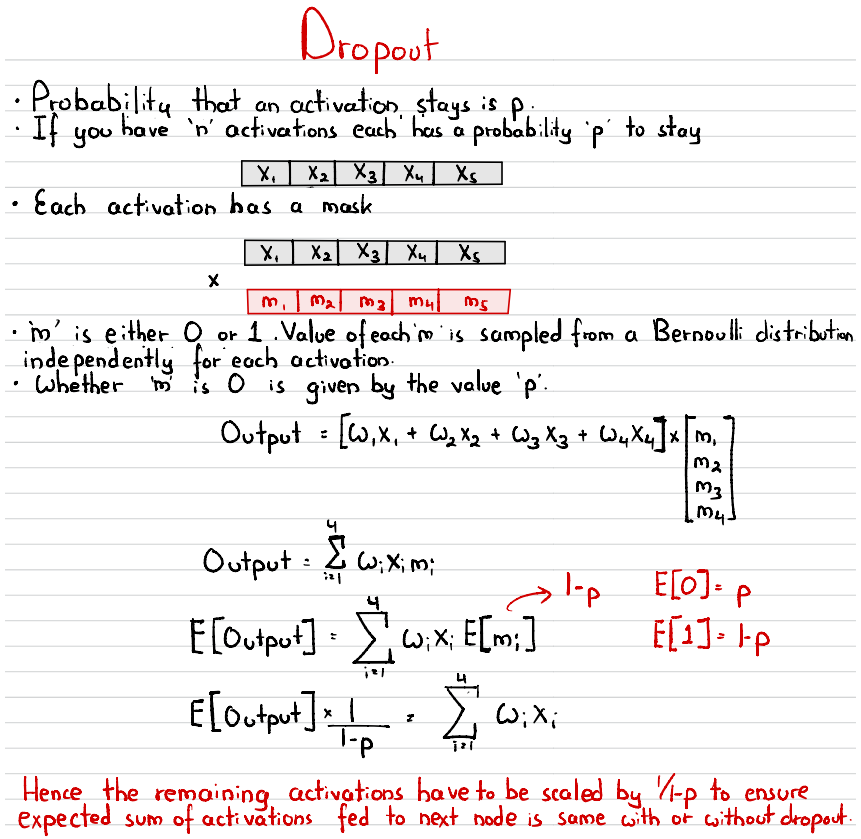
2. Motivation

A motivation for dropout comes from a theory of the role of sex in evolution (Livnat et al., 2010). Sexual reproduction involves taking half the genes of one parent and half of the other, adding a very small amount of random mutation, and combining them to produce an offspring. The asexual alternative is to create an offspring with a slightly mutated copy of the parent's genes. It seems plausible that asexual reproduction should be a better way to optimize individual fitness because a good set of genes that have come to work well together can be passed on directly to the offspring. On the other hand, sexual reproduction is likely to break up these co-adapted sets of genes, especially if these sets are large and, intuitively, this should decrease the fitness of organisms that have already evolved complicated co-adaptations. However, sexual reproduction is the way most advanced organisms evolved.

One possible explanation for the superiority of sexual reproduction is that, over the long term, the criterion for natural selection may not be individual fitness but rather mix-ability of genes. The ability of a set of genes to be able to work well with another random set of genes makes them more robust. Since a gene cannot rely on a large set of partners to be present at all times, it must learn to do something useful on its own or in collaboration with a *small* number of other genes. According to this theory, the role of sexual reproduction is not just to allow useful new genes to spread throughout the population, but also to facilitate this process by reducing complex co-adaptations that would reduce the chance of a new gene improving the fitness of an individual. Similarly, each hidden unit in a neural network trained with dropout must learn to work with a randomly chosen sample of other units. This should make each hidden unit more robust and drive it towards creating useful features on its own without relying on other hidden units to correct its mistakes. However, the hidden units within a layer will still learn to do different things from each other. One might imagine that the net would become robust against dropout by making many copies of each hidden unit, but this is a poor solution for exactly the same reason as replica codes are a poor way to deal with a noisy channel.

A closely related, but slightly different motivation for dropout comes from thinking about successful conspiracies. Ten conspiracies each involving five people is probably a better way to create havoc than one big conspiracy that requires fifty people to all play their parts correctly. If conditions do not change and there is plenty of time for rehearsal, a big conspiracy can work well, but with non-stationary conditions, the smaller the conspiracy the greater its chance of still working. Complex co-adaptations can be trained to work well on a training set, but on novel test data they are far more likely to fail than multiple simpler co-adaptations that achieve the same thing.

Dropout is more effective than other regularization such as weight decay (L2 reg), filter norm constraints and sparse activity regulariazation. (Srivastava et. al. 2014)



Spatial Dropout and Gaussian dropout

Zero Shot Learning, Generalized Zero Shot Learning, Few Shot Learning

- In zero shot learning during test time learner observes samples not viewed during training and needs to predict the class. Works by associating observed and non observed classes through auxiliary information which encodes distinguishing properties of the objects. So in addition to providing image, dataset also contains some numeric/ categorical attributes pertaining to that image. For example in case of animals some of these attributes could be color, skin features (stipes, fur etc.), habitat, size etc. A network trained only on horse images and attributes would be able to identify a zebra.
- While zero shot learning aims to improve performance only on the test set. Generalized zero shot learning aims to improve performance on both train and test sets.
- Few shot learning is another learning paradigm which aims to improve generalization capabilities given few training examples.

Visualizing the Loss Function landscapes [1]

- The trainability of neural nets is highly dependent on factors such as network architecture, weight initialization, choice of optimizer etc.
- How does loss landscape geometry affect generalization error and trainability ?
- As the networks become more deep, loss function landscapes quickly transition from convex to chaotic behavior.
- Skip connections promote flat minimizers and prevent transition to chaotic behavior.
- **Influence of network architecture on the loss landscape :**
  - **Depth influence :** Figure 5 top row shows ResNet architectures (contain skip connections) and bottom row is VGG architectures which do not have skip connections. As you move from left to right the network depth increases and so does the complexity of the loss landscape about the minimizer when no skip connections are used.
  - **Width Influence :** Figure 6 shows the effect of increasing the width of the network architectures (in this case the number of filters). With skip connections (top row) increasing the width of the network leads to more flatter and less chaotic loss landscapes. Similar with no skip connections (bottom row)
  - **Flatter minimizers lead to lower test errors.**
  - In the case with no skip connections, increasing the network width has the opposite affect on the flatness of the loss landscape compared to increasing the network depth.
- **Influence of weight initialization scheme on the loss landscape :**
  - Figure 5 show that regions of convexity and chaotic regions seem to be well partitioned.
  - When using normalized random weight initialization schemes as proposed by Glorot and Bengio (Glorot Normaliztion) typical neural networks achieve an initial loss value less than 2.5. Well behaved loss landscapes in ResNet and shallow VGG are mostly dominated by large, flat and nearly convex attractors that rise to a loss value of greater than 4. For such landscapes, a random initialization would likely lie in the “well behaved” loss region and the optimiztaion algorithm might never see the non convexities on the high loss chaotic plateaus.

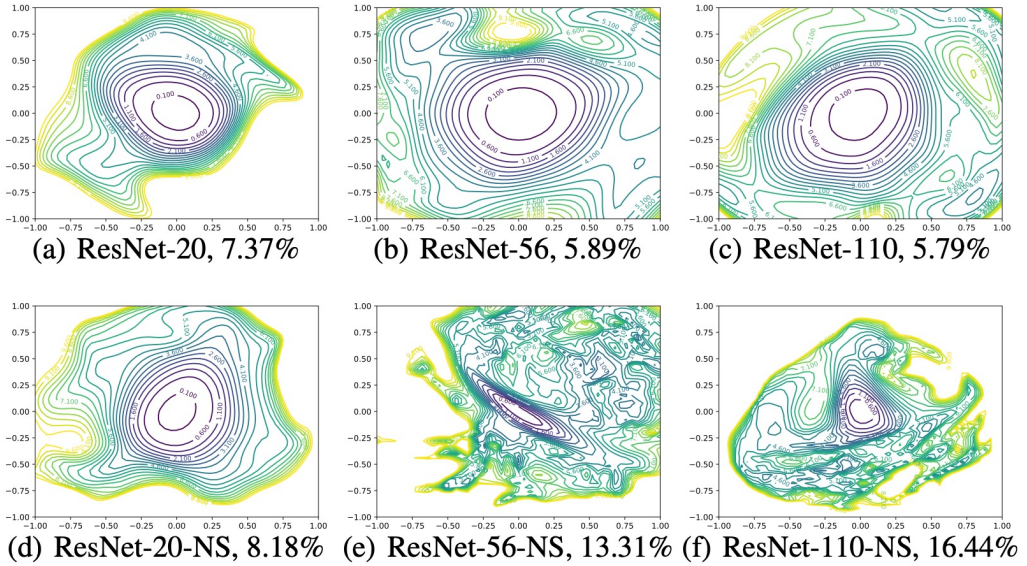


Figure 5: 2D visualization of the loss surface of ResNet and ResNet-noshort with different depth.

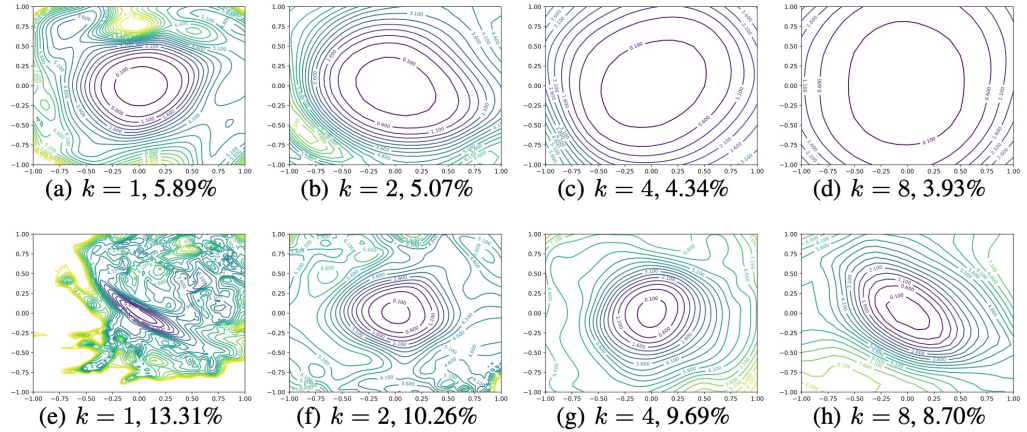


Figure 6: Wide-ResNet-56 on CIFAR-10 both with shortcut connections (top) and without (bottom). The label $k = 2$ means twice as many filters per layer. Test error is reported below each figure.

Are we really seeing convexity ?

- If non convexity present in dimensionality reduced plot then also present in full d surface also. Apparent convexity in low d plot does not mean high d plot is truly convex rather ist means positive curvatures are dominant

[1] Li, H.; Xu, Z.; Taylor, G.; Studer, C.; Goldstein, T. Visualizing the Loss Landscape of Neural Nets. 11.

Nesterov Momentum

- Stochastic gradient descent updates a weight 'w' as follows :

$$w_{t+1}^i = w_t^i - \alpha \nabla l(w_t^i)$$

- Stochastic gradient descent with momentum adds an additional component called velocity to SGD as follows : (α learning rate and β is momentum)

$$\vartheta_{t+1}^i = \left(\beta \vartheta_t^i\right) + \left(-\alpha \nabla l\left(w_t^i\right)\right) \text{ (Update amount calculation)}$$

$$w_{t+1}^i = w_t^i + \vartheta_{t+1}^i \text{ (Applying update)}$$

- If $\vartheta_{t+1}^i = -\alpha \nabla l(w)$ then we essentially get vanilla SGD. Both $\beta \vartheta_t^i$ and $-\alpha \nabla l(w)$ are weight update terms. α and β are like scalar multiplier controlling the length of the 2 weight update vectors. The term $\beta \vartheta_t^i$ is the momentum term that takes the previous update value ϑ_t^i , multiplies it by a factor of β which is the momentum factor and the adds it the current gradient update. It is meant to speed up movement along directions of strong loss decrease. If the previous gradient update is pointing towards direction of large loss decrease then we want to compound that to the current gradient update so that we can accelerate along the direction of large loss decrease. Another advantage is that it also prevents getting stuck in local minima because the momentum gained enables you to jump out of that local optimum point.
- But what if the momentum vector points in a direction different than the what the gradient is telling to move in. If the momentum update term over powers the gradient update term then there is a possibility we could be moving in the wrong direction. One way to deal with this situation is instead of doing gradient descent on the current w_t^i we do an intermediate update where we add $\beta \vartheta_t^i$ to w_t^i. Now if the addition of $\beta \vartheta_t^i$ results in an undesirable increase in the objective then applying gradient descent on ($w_t^i + \beta \vartheta_t^i$) should point back towards w_t^i. Hence this prevents moving in an inappropriate direction dictated by momentum.

Ref : http://proceedings.mlr.press/v28/sutskever13.pdf

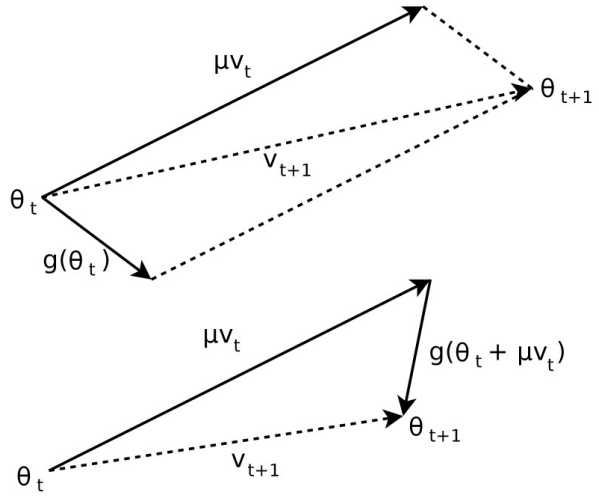


Figure 1. **(Top)** Classical Momentum **(Bottom)** Nesterov Accelerated Gradient

Check where the momentum is pointing you to

$$\vartheta_{t+1}^i = \left(\beta \vartheta_t^i\right) + \left(-\alpha \nabla l\left(w_t^i + \beta \vartheta_t^i\right)\right) \text{ (Update amount calculation)}$$

Apply gradient correction on projected location

Accelerate movement along this corrected direction

$w_{t+1}^i = w_t^i + \vartheta_{t+1}^i$ (Now finally apply the update to the weight)

Adam optimizer

- Adamax is a variant of Adam based on the infinity norm. Nadam is a variant of Adam including Nesterov momentum.
- The 'Ad' in Adam stands for adaptive estimation and 'm' stands for moment which is of first and second order. Therefore Adam does adaptive moment estimation.
- It calculates individual adaptive learning rates for each parameter based on estimation of first and second order moments of the gradients.
- Combines AdaGrad which works well with sparse gradients and RMSProp.
- Consider f(w) as the objective parameterized by parameter set w. We are interested in minimizing E[f(w)] wrt to w where at each instance there is different realization of the function ($f_1(w)$, $f_2(w)$ …) which occurs due to evaluation at random minibatches. For example we are trying to minimize MSE loss function which is ($y_{true} - y_{pred}$)^2 where y_{pred} is parameterized by the weights and biases of the model for which we want to find the appr weights and biases. y_{true} varies between samples or minibatches givign rise to the stochasticity. $g_t = \nabla_w f(w)$.
- Algorithm updates exponential moving averages of the gradient m_t and squared gradient v_t where the hyper params β_1 and β_2 control the exponential decay rates of these moving averages.
- Moving averages are estimates of the 1st moment (mean) and 2nd raw moment (uncentered variance) of the gradient. These moving averages are initialized with 0's leading to moment estimates that are biased towards 0 especially when $\beta's$ close to 1. The initialization bias can be counteracted resulting in bias corrected estimates of 1st and 2nd moments of the gradient denoted as $\hat{m}_t$ and $\hat{v}_t$ respectively.
- The update applied to weight is of the following format : (ϵ added for numerical stability takes vales of 1e-8)

$$w_{t+1}^i = w_t^i - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \quad \text{(Applying update)}$$

- **An important point to note is that although you do not have to try and search for an exact value of the learning rate, the order of magnitude (ex : 1e-2, 1e-3, 1e-4, 1e-5) does matter when using these adaptive learning rate optimizers. The magnitude of steps taken in the parameter space 'w' is bounded by the learning rate magnitude you choose. Update amount <= α. This establishes a trust region around the chosen parameter value.**
- **Step size also invariant to gradient scaling.**

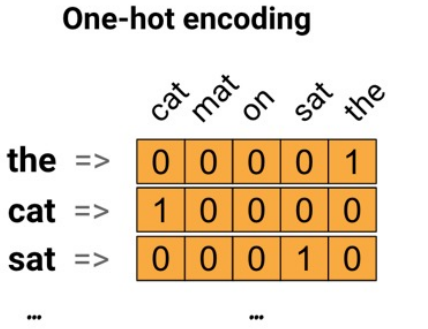
Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1st moment vector)
$v_0 \leftarrow 0$ (Initialize 2nd moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

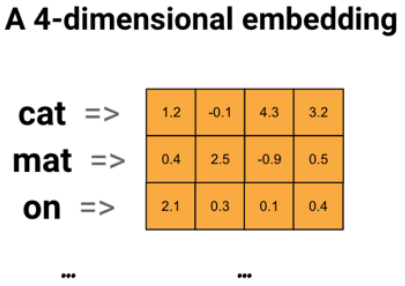
Ref : https://arxiv.org/pdf/1412.6980.pdf

- Depth wise convolution applies a different kernel for each of the input channels unlike in regular convolution layers where the same kernel is applied to each of the channels.

Word embeddings



- This is an alternative to categorical data where you would typically use one hot encoding. Here we vectorize or tokenize the entries.
- For ex : The cat sat on the mat. Unique words in this sentence are (The, cat, sat, on, the, mat). A one hot encoding of each word would look something on the side. To represent the sentence you would concatenate the one hot encoded vectors. Dosadvnatges with one hot encoding is its inherent sparsity as the number of classes start to increase.
- Method 2 : Encode each word with a unique number
  - The integer-encoding is arbitrary (it does not capture any relationship between words).
  - An integer-encoding can be challenging for a model to interpret. A linear classifier, for example, learns a single weight for each feature. Because there is no relationship between the similarity of any two words and the similarity of their encodings, this feature-weight combination is not meaningful.
- Method 3 : Use a dense representation in which similar words have a similar encoding.
  - An embedding is a dense vector of floating point values. Length of vector is a parameter you specify.
  - Instead of specifying the values of the mebedding manually, they are trainable parameters.
  - Common to see word embeddings that are 8 dimensional (for small datasets) and upto 1024 dimensional (for large datasets) A higher dimensional embedding can capture fine-grained relationships between words, but takes more data to learn.



- Ref : https://www.tensorflow.org/text/guide/word_embeddings?hl=en

- Apply a gaussian prior on the latent. L1 norm on middle reconstructions and L2 on final
- Use a different weight initialization scheme
- Try this idea create a very low res of the image that only capturesthe position and keep on refining the image thorugh upsampling and conv layers. Try using a 4x4 or 8x8 pixel image, check with training image and then try upscaling from there. Remove the res connections and try with a very simple design first and then start adding more details to it.

- Additional changes made :
  - Converted an intermediate Resizing layer to Upsampling2D layer.
  - Made all interpolation to “Nearest”

Proper weight and Bias initialization

Weight and Bias initialization

- In training neural networks we want to minimize the loss.
- The loss function can be any type of function that is continuous and differentiable.
- Consider the MSE function

$$L = (y - \hat{y})^2$$

- In some cases we have a closed form solution to the optimization problem.

$L = (y - \omega_1 x)^2$ (Single node, Single layer NN)

$\frac{\partial L}{\partial \omega_1} = -2(y - \omega_1 x)x = 0$

$x \neq 0 \;\therefore\; \omega_1 = \frac{y}{x}$

$L = x^2 + y^2$

$\frac{\partial L}{\partial x} = 2x = 0 \quad \therefore x = 0 \qquad \frac{\partial L}{\partial y} = 2y = 0 \;\therefore y = 0$

$(x, y) = (0, 0)$ is the minima

* In multivariable functions, finding the partial derivatives wrt each variable and setting them to 0 will give a set of coupled Eqns to solve

- In cases where the closed form of solution to the optimization problem cannot be found analytically we have to rely on numerical methods. Most famous one is Newton Raphson.
- Deriving Newton Raphson for maximizing a function with one random variable is straight forward

Taylor Series expansion of $f(x+h)$

$$f(x+h) \approx f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 \cdots \quad \text{(Second Order Taylor)}$$

To find the value of 'h' that maximizes $f(x+h)$, we need to take derivative of function wrt h and equate to 0. Consider $\hat{h}$ to be the value that maximizes the function.

$$f'(x+\hat{h}) \approx f'(x) + f''(x)\hat{h} = 0$$

$$\hat{h} = -\frac{f'(x)}{f''(x)}$$

$$x + h = x - \frac{f'(x)}{f''(x)}$$

$$x_{new} = x_{old} - \frac{f'(x_{old})}{f''(x_{old})}$$

* NR does not check if $f''(x_{new}) < 0$ to ensure x_{new} is a maxima. Hence if your initialization is bad you can end up at a minima rather than maxima.

- Now lets look at gradient descent and all algorithms based on it. Gradient descent is the default go to algorithm for Machine Learning.

$$W = W - \alpha dW \quad \text{(Stochastic Gradient Descent)}$$

If found a large neg. gradient then keep moving in that direction

$$\left\{ \begin{array}{l} V_{dw} = \beta V_{dw} + (1-\beta) dW \\ W = W - \alpha V_{dw} \end{array} \right\} \quad \text{(Momentum)}$$

This is still Gradient Descent Same lr for all params

This is similar to momentum

$$\left\{ \begin{array}{l} S_{dw} = \beta S_{dw} + (1-\beta) dw^2 \\ W = W - \frac{\alpha}{\sqrt{S_{dw}} + \epsilon} dw \end{array} \right\} \quad \text{(RMSProp)}$$

kind of like LR annealing. Whats interesting is that this is per param lr

This is our momentum equation

$$\left\{ \begin{array}{l} V_{dw} = \beta_1 V_{dw} + (1-\beta_1) dw \\ S_{dw} = \beta_2 S_{dw} + (1-\beta_2) dw^2 \\ V_{corr_{dw}} = \frac{V_{dw}}{(1-\beta_1)^t} \qquad S_{corr_{dw}} = \frac{S_{dw}}{(1-\beta_2)^t} \\ w = w + \frac{\alpha}{\sqrt{S_{corr_{dw}}} + \epsilon} V_{corr_{dw}} \end{array} \right\} \quad \text{(Adam)}$$

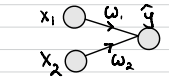
Per param lr annealing

There are 3 main components to the gradient descent algorithm
1. Choice of initial parameters (weights and bias)
2. Learning rate
3. Batch size

In this tutorial we will look at different weight initialization schemes.

Zero and Constant Initialization

- Consider a 2 layer NN with $b=0$ and $\omega_1 = \omega_2 = 0$



$\hat{y} = \omega_1 x_1 + \omega_2 x_2$

$L = (y - \hat{y})^2$

$\frac{dL}{d\omega_1} = -2(y-\hat{y})x_1 \qquad \frac{dL}{d\omega_2} = -2(y-\hat{y})x_2$

Now consider initializing $\omega_1 = \omega_2 = 0$ and find new ω_i through SGD.

$$\omega_1^{t+1} = \omega_1^t - 2\alpha(y-\hat{y})x_1 \qquad \omega_2^{t+1} = \omega_2^t - 2\alpha(y-\hat{y})x_2$$

$$\omega_1^{t+1} = 0 - 2\alpha y x_1 \qquad \omega_2^{t+1} = -2\alpha y x_2$$

What we see is a symmetry in our weights due to identical gradients. Weights will evolve symmetrically throughout training.

- Now consider same case of $b=0$ but $\omega_1 = \omega_2 = \alpha$

$$\omega_1^{t+1} = \alpha - 2\alpha(y - \alpha(x_1+x_2))x_1 \qquad \omega_2^{t+1} = \alpha - 2\alpha(y - \alpha(x_1+x_2))x_2$$

We again observe a symmetry in our weights. Hence constant/zero initialization is not useful for learning.

Glorot / Xavier Initialization

Goal: Find relation b/w $Var[a^l]$ and $Var[a^{l-1}]$

(Exp with 2 layer net on ped)

$z^{l-1} = \omega^{l-1} a^{l-2} + b^{l-1}$

$a^{l-1} = \tanh(z^{l-1})$

if mean and activation of $f(y)$ outputs from each layer is same we will not get exp or vanish grad.

$z^l = \omega^l a^{l-1} + b^l$

$a^l = \tanh(z^l)$

In the early phases of training we are in the linear regime of tanh. $\tanh(x) \approx x$

$$Var[a^l] \approx Var[z^l]$$

$$Var[a^l] \approx Var[\omega^l a^{l-1} + b^l]$$

Assume $b = 0$

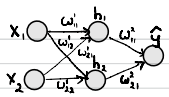
$$Var[a^l] \approx Var[\omega^l a^{l-1}] + Var[b^l]$$

Essentially means ω_1^2 and ω_2^2 are ind. and are drawn from the same dist.

Assumptions:
1. Weights are {independent and identically distributed} (Symmetric Gaussian with only diag. covariance)
2. Inputs are independent and identically distributed
3. Weights and inputs are mutually independent.
4. Linear regime of tanh.

[1] https://www.deeplearning.ai/ai-notes/initialization/index.html

[2] https://wandb.ai/site/articles/the-effects-of-weight-initialization-on-neural-nets



$L = (y-\hat{y})^2 \quad \frac{dL}{d\hat{y}} = 2(y-\hat{y}) \quad \frac{d\hat{y}}{d\omega_{11}^2} = h_1 \quad \frac{d\hat{y}}{d\omega_{21}^2} = h_2$ (Assume linear act.) ($\hat{y} = h_1\omega_{11}^2 + h_2\omega_{21}^2$)

$\omega_{11}^2 = \omega_{11}^2 + \alpha\frac{dL}{d\omega_{11}^2} = \omega_{11}^2 + \alpha\frac{dL}{d\hat{y}}\frac{d\hat{y}}{d\omega_{11}^2} = \omega_{11}^2 + \alpha 2(y-\hat{y})h_1$

($h_1 = \omega_{11}^1 x_1 + \omega_{21}^1 x_2$)

$\omega_{11}^1 = \omega_{11}^1 + \alpha\frac{dL}{d\omega_{11}^1} = \omega_{11}^1 + \alpha\frac{dL}{d\hat{y}}\frac{d\hat{y}}{dh_1}\frac{dh_1}{\omega_{11}^1} = \omega_{11}^1 + 2\alpha(y-\hat{y})\omega_{11}^2(x_1)$

We can see that weights ω_{11}^2 and ω_{11}^1 depend on the activations h_1 and x_1 respectively. Hence if we ensure the mean and variance of our activations is some across layers we can prevent exploding or vanishing gradients

$$Var[a^l] = Var[a_k^l]$$ Each of these activations is a Random variable

* Variance of vector of activations is same as variance of each activation since we assume each activation is independent and drawn from the same distribution (Assump 2)

$$Var[a_k^l] = Var[z_k^l] = Var\left[\sum \omega_{kj}^l a_j^{l-1}\right] = \sum Var(\omega_{kj}^l a_j^{l-1})$$ (Assump 3)

① $E[X] = \sum p_i x_i$

② $Var[X] = E[(X - E(X))^2] = E[X^2 - 2XE(X) + (E(X))^2] = E[X^2] - 2E[X]E[X] + (E[X])^2$

$= E[X^2] - (E[X])^2$

③ $Var[X+Y] = E[X^2 + 2XY + Y^2] - (E[X+Y])^2$

$= E[X^2] + 2E[XY] + E[Y^2] - (E[X])^2 - 2E[X]E[Y] + (E[Y])^2$

$= Var[X] + Var[Y]$ (only if X and Y are independent RV)

④ $Var[X_1Y_1 + X_2Y_2 \cdots X_nY_n] = Var[X_1Y_1] + Var[X_2Y_2] \cdots Var[X_nY_n]$ (Assump 3)

⑤ $Var[X_1Y_1] = (E[X_1])^2 Var[Y_1] + (E[Y_1])^2 Var[X_1] + Var[X_1]Var[Y_1]$

$Var[\omega_{kj}^l a_j^{l-1}] = (E[\omega_{kj}^l])^2 Var[a_j^{l-1}] + (E[a_j^{l-1}])^2 Var[\omega_{kj}^l]$

$+ Var[\omega_{kj}^l]Var[a_j^{l-1}]$

Assume both mean of weights and activations centered abt 0.

$Var[a_k^l] = \sum Var(\omega_{kj}^l a_j^{l-1}) = \sum Var[\omega_{kj}^l]Var[a_j^{l-1}]$

$= n^{l-1} Var[\omega_{kj}^l]Var[a_j^{l-1}]$

$\therefore n^{l-1} Var[\omega_{kj}^l] = 1$ or $Var[\omega_{kj}^l] = 1/n^{l-1}$

- From forward propagation $Var[\omega_{kj}^l] = 1/n^{l-1}$ obtained from constraint $Var[a_k^l] = Var[a_j^{l-1}]$
- From backward propagation $Var[\omega_{kj}^l] = 1/n^l$ obtained from constraint $Var\left[\frac{dL}{d\omega_{kj}^l}\right] = Var\left[\frac{dL}{d\omega_{j+1}^l}\right]$
- As comprise b/w 2 constraints we use $Var[\omega_{kj}^l] = \frac{2}{n^{l-1} + n^l}$
- Above derived for Xavier Normal for Xavier Uniform $\frac{6}{n^{l-1} + n^l}$
- Xavier works only with linear activations or symmetric activation fns like tanh or softsign. If using non symmetric activation like ReLU use He initialization
- Also Xavier requires activations to be independent and identically distributed.

He initialization

Assumptions

① Weights are independent and identically distributed ④ $E[\omega_{kj}^l] = E[a_j^{l-1}] = 0$
② Activations are independent and identically distributed and linear
③ Weights and activations are mutually independent.

$$Var[a_k^l] = Var[z_k^l] = Var\left[\sum_1^{n^{l-1}} \omega_{kj}^l a_j^{l-1}\right]$$ (Assumpt act is linear)

$$Var[a_k^l] = \sum_1^{n^{l-1}} Var(\omega_{kj}^l a_j^{l-1})$$ (Wts and acts are ind.)

$$= \sum_1^{n^{l-1}} E[(\omega_{kj}^l a_j^{l-1})^2] - (E[\omega_{kj}^l a_j^{l-1}])^2$$ (Assump 3) ↳ $E[\omega_{kj}^l]E[a_j^{l-1}]$

$$= \sum_1^{n^{l-1}} E[(\omega_{kj}^l)^2 (a_j^{l-1})^2]$$

$$= \sum_1^{n^{l-1}} Var[\omega_{kj}^l] E[(a_j^{l-1})^2]$$ (Assump 3)

$$= n^{l-1} Var[\omega_{kj}^l] E[(a_j^{l-1})^2]$$

$E[(a_j^{l-1})^2] = \frac{1}{2} Var[a_j^{l-1}] \quad \therefore Var[a_k^l] = \frac{n^{l-2}}{2} Var[\omega_{kj}^l]Var[a_j^{l-1}]$

$\therefore Var[\omega_{kj}^l] = \frac{2}{n^{l-2}}$

Generally this problem of vanishing and exploding gradients occurs in deep stacked FFNN's.

Table 11-1. Initialization parameters for each type of activation function

Initialization	Activation functions	σ^2 (Normal)
Glorot	None, tanh, logistic, softmax	$1 / fan_{avg}$
He	ReLU and variants	$2 / fan_{in}$
LeCun	SELU	$1 / fan_{in}$

(1) Géron, A. Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow. 1150.

Choice of Activation functions

- **Sigmoid** $f(x) = \frac{1}{1 + e^{-x}}$ $f'(x) = \frac{e^{-x}}{(1 + e^{-x})^2}$

  Problem : Vanishing gradients in large NN. Gradient max at x=0 equal to 0.25.

- **tanh** $f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$ $f'(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$

  Problem : Vanishing gradients in large NN. Gradient max at x=0 equal to 1.

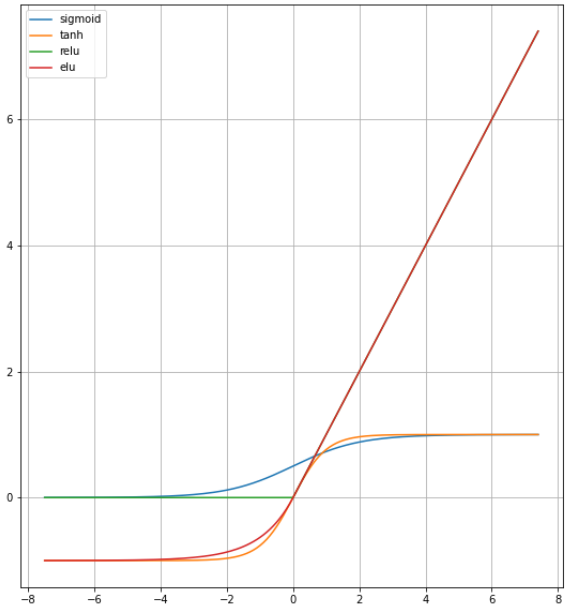
- **ReLU** $f(x) = \begin{cases} x & x > 0 \\ 0 & x \leq 0 \end{cases}$ $f'(x) = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases}$

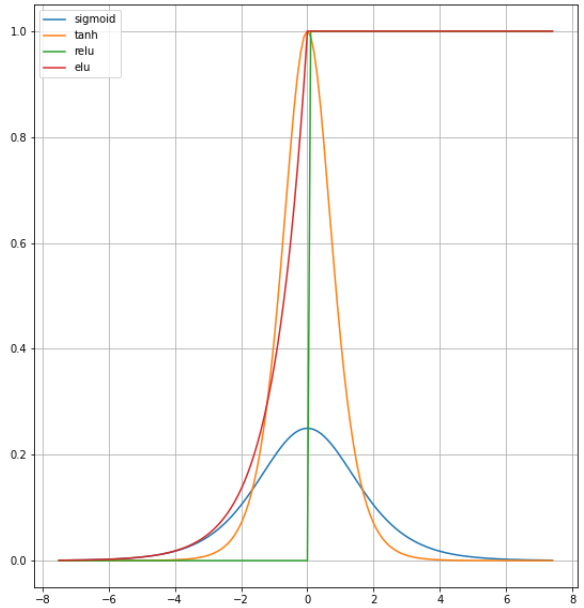
  Problem : Dead units when input less than 0. Grad signal also is 0.

- **ELU and SELU (applies a scaling factor to ELU)**

  $f(x) = \begin{cases} x & x > 0 \\ e^{x} - 1 & x \leq 0 \end{cases}$ $f'(x) = \begin{cases} 1 & x > 0 \\ e^{x} & x \leq 0 \end{cases}$

- **Softmax** $f(x) = \frac{e^{x}}{\sum e^{x}}$ $f'(x) = \frac{e^{x}}{\sum e^{x}}$





ELU activation function :

- RELU is a linear function for positive values and 0 for negative values this causes the mean of the activations after each hidden layer to shift from 0.
- Unit that have non zero activation have a bias for the next hidden layer. This creates a bias shift.