Machine Learning and Pattern Recognition - Week 8 - Autoencoders & PCA

Antonio León Villares

November 2022

Contents

1 Unsupervised Representation Learning: Autoencoders **2**
1.1 Unsupervised Representation Learning 2
1.2 Autoencoders 2

2 Unsupervised Representation Learning: PCA **5**
2.1 Covariance Matrices 5
2.2 Principal Component Analysis 7
2.3 Alternative Decomposition: SVD 10
2.4 Probabilistic PCA 11

3 Question **12**
3.1 Notes Questions 12

4 Tutorial **12**

Based on the online notes here.

1 Unsupervised Representation Learning: Autoencoders

1.1 Unsupervised Representation Learning

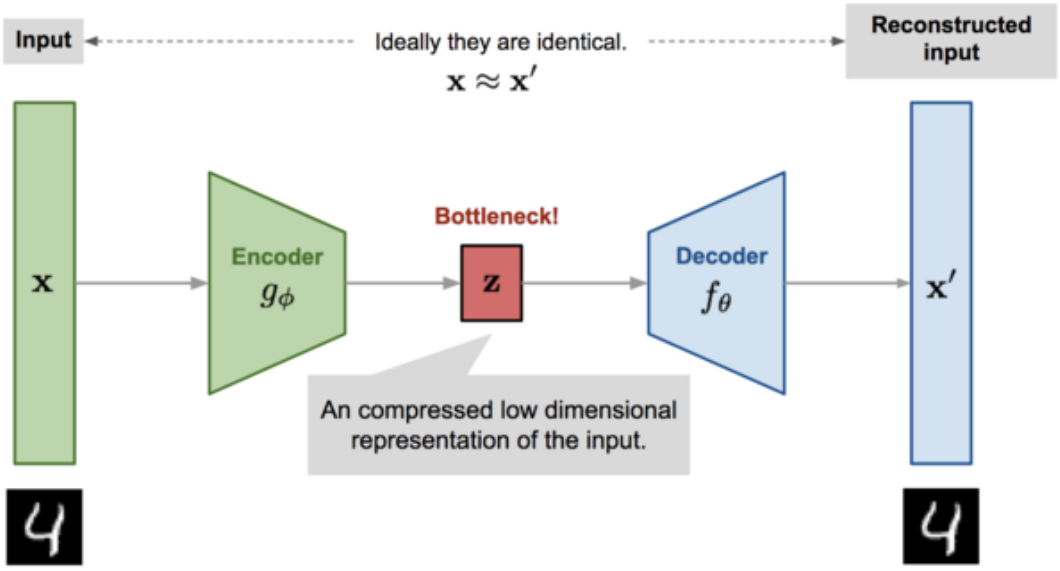
- **How do neural networks help identify features in our data?**
  - given **targets**, a **neural network** learns **weights** which convert an **input** into a **feature vector**
  - this is **supervised representation learning**
- **Why is unsupervised representation learning important?**
  - **good labelled** data is **scarce**
  - **unsupervised representation learning** seeks to **learn** features for **unlabelled data**
  - these representation are useful, since:
    1. They provide **features** which are **inherent** in the data, allowing us to better **understand** the data (i.e given an image, it can learn that edges are important for differentiating between 2 inputs)
    2. They allow us to **reduce** the **dimensionality** of the data, so:
       * computations can be more **efficient**
       * we can **visualise** high dimensional data
       * useful for **downstream** tasks (i.e clustering, ourlier detection)

1.2 Autoencoders

- **What is an autoencoder?**
  - a **neural network architecture** which is **self-supervised**
  - it seeks to learn representations of data by learning the **identity function**:

$$f(\underline{x}) = \underline{x}$$

  - in order to learn a **useful** representation, we typically place **constraints** on the architecture
- **How does the data processing inequality affect the power of autoencoder?**
  - an **autoencoder** learns a **new** representation of the data which we feed
  - however, it can't **add information** to this representation: it will either **lose** or **keep** the information
  - as such, **autoencoder constraints** must be **carefully** chosen, to ensure we truly extract the **meaningful features**
- **How does a bottleneck autoencoder learn features?**
  - with a **bottleneck constraint**, if we have an input $\underline{x} \in \mathbb{R}^D$, we require that the **hidden representation** is **lower-dimensional**, $\underline{h} \in \mathbb{R}^K$, where $K << D$:



  - by doing this, we **force** the network to learn a **meaningful, compressed** representation of the input
  - the more meaningful the features it finds, the easier the reconstruction will be

- **How does a denoising autoencoder learn features?**
  - a **denoising autoencoder** doesn't require learning a **compressed representation**
  - input features are **corrupted**, by randomly setting certain values to 0
  - the autoencoder then has to **denoise** the input, to reconstruct the original input

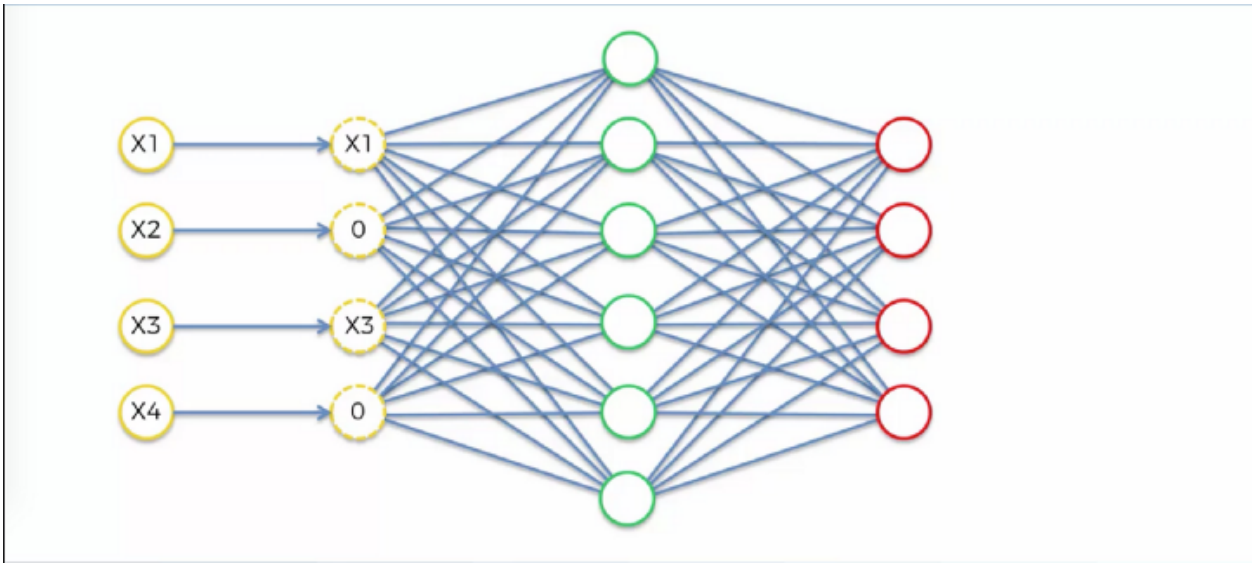


Figure 1: The denoising autoencoder learns a larger feature representation $D \geq K$ to denoise the corrupted input.

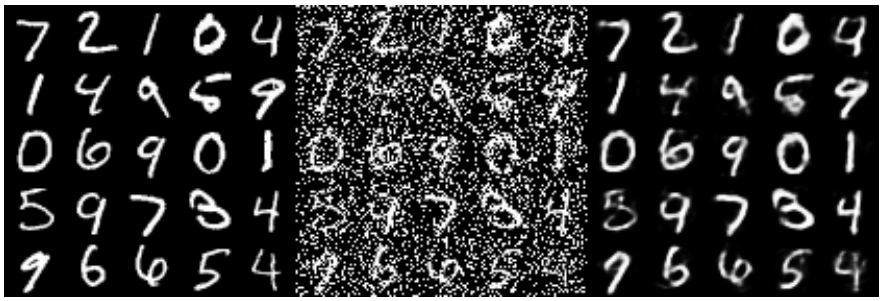
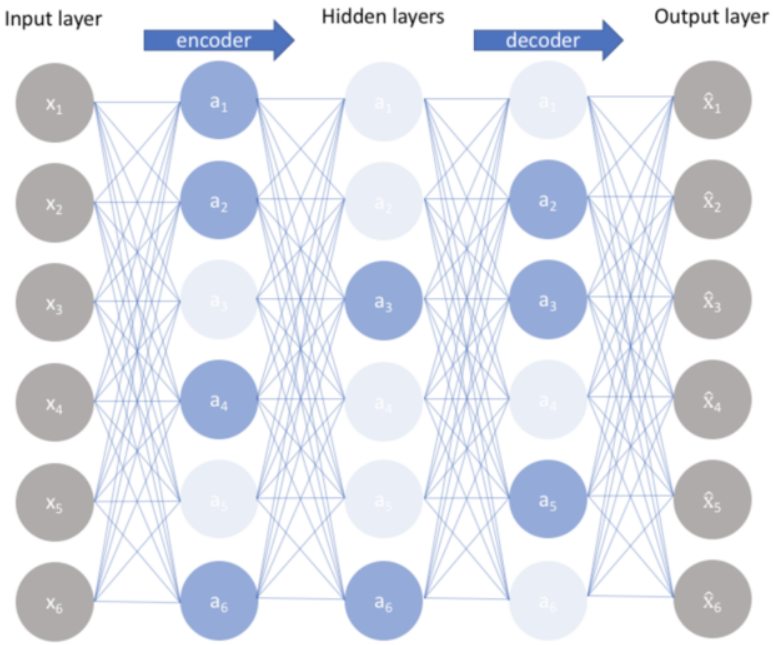


Figure 2: Example of corrupted data, and how the denoising autoencoder removes the noise.

  - the idea is that the autoencoder will learn to **combine** the non-zero features to **reconstruct** the **missing features**; by doing this, it is learning the underlying structures which define an object, even if there is noise involved

- **How does a sparse autoencoder learn features?**
  - a **sparse autoencoder** limits the amount of **hidden units** which can be non-zero (that is, it enforces a large proportion of hidden units to be set to 0)



  - this architecture forces the network to be "intelligent" about how a select number of features can be combined to reconstruct the input
  - thus, even if we have a large number of input features, the network learns to only rely on a few of them - the ones which are **meaningful** and **truly representative**
  - more details of implementation can be found in these notes by Andrew Ng

- **Why is it important to set constraints on the autoencoder architecture?**
  - our objective is that $f(\underline{x}) \approx \underline{x}$

– without constraints, a network can learn:

$$f(\underline{x}) = W\underline{x}$$

where $W = \mathbb{I}$

– this certainly fulfills the objective $f(\underline{x}) \approx \underline{x}$, but it isn't learning a **useful representation**

2 Unsupervised Representation Learning: PCA

2.1 Covariance Matrices

- **How can we use an eigendecomposition to sample from a multivariate normal distribution?**
  - say we have a **multivariate Gaussian**:

$$\mathcal{N}(\underline{0}, \Sigma)$$

  - by the **Spectral Theorem**, since Σ is **real** and **diagonal**, it has an **eigendecomposition** involving only **real** eigenvalues, with **orthonormal** eigenvectors
  - moreover, we can then **decompose** Σ via:

$$\Sigma = Q\Lambda Q^T$$

where:

* Q is an **orthogonal** matrix, with columns as the **eigenvectors** of Σ
* Λ is a **diagonal** matrix, with the **eigenvalues** of Σ as the diagonal elements

*Let $A \in \mathbb{R}^{n\times n}$ be a matrix. An **eigenvalue** of A is a constant $\lambda \in \mathbb{C}$, such that $\exists \underline{v} \in \mathbb{R}^n$ satisfying:*

$$A\underline{v} = \lambda\underline{v}$$

*In particular, if we want to compute λ, will be the **roots** of the **characteristic polynomial**:*

$$P(\lambda) = |A - \lambda\mathbb{I}| = 0$$

*where $|\cdot|$ is the **determinant**.*
If $n = 2$ and:

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

then:

$$P(\lambda) = (a-\lambda)(d-\lambda) - bc = \lambda^2 - \lambda(ad) + (ad - bc)$$

which has solutions:

$$\lambda = \frac{ad \pm \sqrt{a^2d^2 - 4(ad-bc)}}{2}$$

– now recall, since we assume that Σ is positive definite, we know that $\exists A$ such that:

$$\Sigma = AA^T$$

which corresponds to the **covariance matrix** of a random variable $\underline{y} \sim \mathcal{N}(\underline{0}, \Sigma)$ given by:

$$\underline{y} = A\underline{x}$$

where $\underline{x} \sim \mathcal{N}(\underline{0}, \mathbb{I})$ is distributed as a **standard normal random variable**

– hence, to sample from $\mathcal{N}(\underline{0}, \Sigma)$, we need to be able to write:

$$\Sigma = Q\Lambda Q^T = AA^T$$

– notice, if λ_i are the eigenvalues of Σ, define:

$$L = diag(\sqrt{\lambda_1}, \ldots, \sqrt{\lambda_n})$$

then:

$$LL^T = L^2 = \Lambda$$

so:

$$\Sigma = Q\Lambda Q^T = QLL^TQ^T = (QL)(QL)^T$$

so if we define $A = QL$, we can sample from $\mathcal{N}(\underline{0}, \Sigma)$ by using the eigendecomposition

- **How does the distribution sampled from $\mathcal{N}(\underline{0}, \mathbb{I})$ differ from the distribution sampled from $\mathcal{N}(\underline{0}, \Sigma)$?**

  – say $\underline{x} \sim \mathcal{N}(\underline{0}, \mathbb{I})$. Then, drawing samples gievs us a **spherical** distribution:

  – if we sample $\underline{y} \sim \mathcal{N}(\underline{0}, \Sigma)$, this is equivalent to transfoming our samples $\underline{x}$ by using $A = QL$, where:

  1. $L = \sqrt{\Lambda}$ is a **diagonal** matrix, so $L\underline{x}$ **stretches** $\underline{x}$, such that x_d becomes $\sqrt{\lambda_d}x_d$
  2. Q corresponds to a **rotation**, such that the $L\underline{x}$ will now be **aligned** with the eigenvector columns

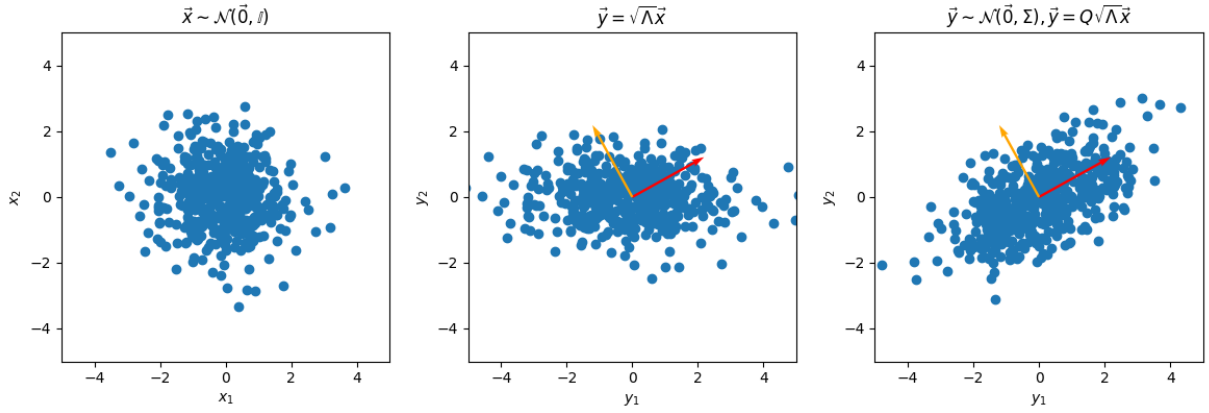


Figure 3: Mapping from $\underline{x}$ to $\underline{y}$, where $\Sigma = \begin{pmatrix} 2 & 0.8 \\ 0.8 & 1 \end{pmatrix}$.

*If you are wondering **why** Q is a rotation, recall its columns are **orthonormal vectors**. As such, they form a **basis** for the space. Moreover, recall, given an orthonormal basis $\underline{v}_i$, if we want to convert a vector $\underline{x}$ described in terms of the standard basis into a vector $\tilde{\underline{x}}$ written in terms of $\underline{v}_i$, we just need to use dot products:*

$$\tilde{\underline{x}}_d = \langle \underline{v}_d, \underline{x} \rangle$$

*Now, matrix multiplication is nothing but a bunch of **dot products**, so when we compute $Q(L\underline{x})$, we are doing nothing but mapping $L\underline{x}$ into its coordinates in terms of the eigenvector basis. Since the eigenvectors are orthonormal, this corresponds to "rotating" our standard x,y axes.)*

2.2 Principal Component Analysis

- **What is principal component analysis?**
  - a **dimensionality reduction** technique
  - can be thought of as a **linear autoencoder** (i.e the non-liner function are set to the identity)
  - in particular, if we want to learn $f(\underline{x}) \approx \underline{x}, \underline{x} \in \mathbb{R}^K$, a **linear** autoencoder will look like:

$$\underline{h} = W^{(1)}\underline{x} + \underline{b}^{(1)}, \qquad W^{(1)} \in \mathbb{R}^{K \times D}$$

$$\underline{f} = W^{(2)}\underline{h} + \underline{b}^{(2)}, \qquad W^{(2)} \in \mathbb{R}^{D \times K}$$

  - with PCA, we **reformulate** this, and seek to find a **single** parameter matrix $V \in \mathbb{R}^{D \times K}$ such that:

$$\underline{h} = W^{(1)}\underline{x} + \underline{b}^{(1)} = V^T(\underline{x} - \bar{\underline{x}})$$

$$\underline{f} = W^{(2)}\underline{h} + \underline{b}^{(2)} = V\underline{h} + \bar{\underline{x}}$$

    where $\bar{\underline{x}}$ is the **mean vector** of our training data:

$$\bar{\underline{x}} = \frac{1}{N}\sum_{n=1}^{N} \underline{x}^{(n)}$$

  - we first center our data by the mean, reduce it to D dimensions, and then we bring it back up to K dimensions with the same parameters, and recenter by adding the mean again

- **How does PCA find the matrix V?**
  - we saw above that the **eigenvectors** of the **covariance matrix** are **orthonormal**, and **span** the whole space
  - when data is distributed with covariance Σ, it will be distributed along an **ellipsoid**, whose **axes** are in the direction of these **eigenvectors**
  - the **length** of these axes will be determined by the **eigenvalues** corresponding to the **eigenvectors**
  - PCA finds the matrix V by **minimising** the **square error** in reconstruction, by minimising the difference between a data point and $\bar{\underline{x}}$ along the **longest** axes of the ellipsoid
  - to do this, it sets V to have as columns the K **eigenvectors** corresponding to the K **largest eigenvalues**

– by doing this, we will be projecting the data into the space spanned by these K eigenvectors
– this minimises **reconstruction error**, since upon reconstructing the data, it will lie alongside the principal components, such that the reconstructed data will be **perpendicular** to the original data

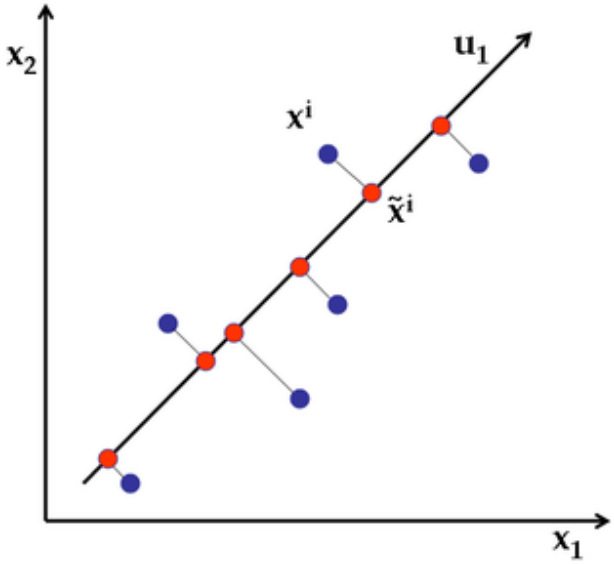


Figure 4: If we reconstruct the data, it will lie along a subspace. By projecting to the principal components (the eigenvectors), data which we reconstruct will lie **perpendicular** to the original data, thus minimising reconstruction error.

– if $K = D$, we use all the eigenvectors, which span the whole space, and thus $VV^T = \mathbb{I}$ (since they are orthonormal)

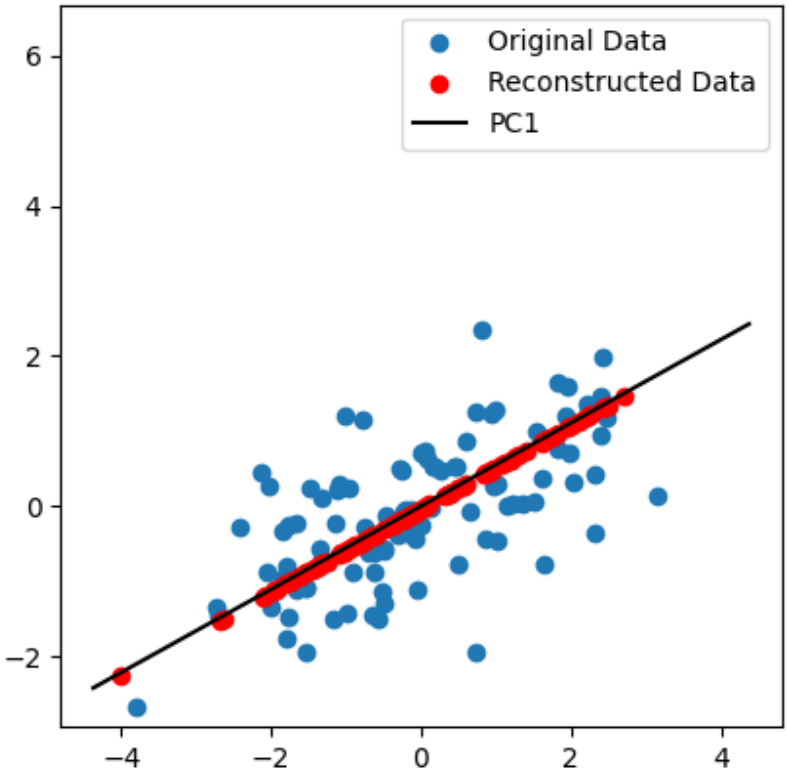


Figure 5: This is the same data as before. We project it down to the first principal component with PCA. If we project back up into 2 dimensions, the data remains on the subspace generated by the eigenvector (a one-dimensional line).

- **What is an alternative interpretation for V?**
  - an alternative viewing of PCA is that by mapping data to the space spanned by the K eigenvectors, we are **maximising** variance of data along these axes
  - maximising the variance ensures thatwe preserve as much of the original information as possible
  - in fact, the eigenvalues provide us with a **proportion** of the variance explained along each axes
- **What advantages does PCA have over gradient-based, non-linear autoencoders?**
  1. **Unique Solution**: V solely depends on **eigenvectors**, which are unique (up to scaling)
  2. **Easy Computation**: computing the parameters V just requires standard linear algebra
  3. **Nesting**: if we choose dimensions K, K^* with $K < K^*$, the first K values in the hidden layer for K and K^* will have the same value (since they both use the same eigenvectors)
- **Does PCA overfit to the data?**
  - technically no: it just uses features of the data to come up with a **sensible** transformation
  - in fact, it can **reduce** overfitting, since it removes the less useful features in the data, which means that models will have less noise to "distract" them

2.3 Alternative Decomposition: SVD

- **What is SVD?**
  - a **matrix decomposition** technique:

$$X = USV^T, \qquad X \in \mathbb{R}^{N \times D}$$

    where:
    - U is an **orthogonal** matrix, with columns given by the **eigenvectors** of $X^T X$
    - S is a **rectangular, diagonal** matrix, where the values in the diagonal are known as **singular values**, and are given by the **square root** of the non-zero eigenvalues of XX^T or $X^T X$
    - V is an **orthogonal** matrix, with columns given by the **eigenvectors** of XX^T

> *To see why, we use the fact that XX^T and $X^T X$ will be real, symmetric matrices, and thus, are **diagonalisable**. Hence:*
>
> $$X^T X = (USV^T)^T (USV^T) = VS^T U^T USV^T = V(S^T S)^T V^T$$
>
> *where we have used the fact that U is **orthogonal** matrix, so $U^T U = \mathbb{I}$. Hence, V must be matrix obtained by taking the eigenvectors of $X^T X$, whilst $S^T S$ must correspond to the eigenvalue matrix of $X^T X$.*
> *Similarly, we get that:*
>
> $$XX^T = (USV^T)(USV^T)^T = USV^T VS^T U^T = U(S^T S)U^T$$
>
> *which we can analyse in a similar way.*

  - this generalises the **eigendecomposition** for non-square matrices

- **What is truncated SVD?**
  - an **approximate** factorisation of X (in fact, the **optimal, low-rank** matrix approximation, in terms of **square error**)
  - instead of using all the eigenvectors/eigenvalues of $X^T X$ and XX^T, we only use a subset
  - in particular:

$$X \approx U_K S_K V_K^T$$

    where:
    - $U_K \in \mathbb{R}^{N \times K}$ contains the K eigenvectors of $X^T X$ corresponding to the K largest singular values
    - $S_K \in \mathbb{R}^{K \times K}$ contains the K largest singular values
    - $V_K \in \mathbb{R}^{D \times K}$ contains the K eigenvectors of XX^T corresponding to the K largest singular values
  - notice, if $K = min(N, D)$, then **truncated SVD** will become SVD, so matrix reconstruction will be perfect

- **How can truncated SVD be used for dimensionality reduction?**
  - the **rows** of U give a K dimensional embedding of the D-dimensional **rows** of X
  - the **columns** of V^T give a K dimensional embedding of the N-dimensional **columns** of X

- hence, truncated SVD gives us **low-dimensional** representations of data, for both **rows** and **columns**, and **all at once**

- **How do PCA and truncated SVD compare?**
  - assume you have **mean-centered** data X
  - then, notice that:
    $$\Sigma = \frac{1}{N} X^T X$$
    since:
    $$\left(\frac{1}{N} X^T X\right)_{ij} = \frac{1}{N} \sum_{k=1}^{n} X_{ki} X_{kj} = \frac{1}{N} \sum_{k=1}^{n} (X_{ki} - 0)(X_{kj} - 0) = \Sigma_{ij}$$
  - hence, matrix U, whose columns are the eigenvectors of $X^T X = \Sigma$ (since $\frac{1}{N}$ is a scaling which doesn't affect the eigenvectors), will be precisely the parameter matrix for PCA
  - similarly, V is built by using the eigenvectors resulting from applying PCA to X^T

2.4 Probabilistic PCA

- **What is the idea behind probabilistic PCA?**
  - if we have very high dimensional data sampled from a distribution, MLE of parameters might be **expensive**
  - with **probabilistic PCA**, we learn to **generate** high-dimensional data, by using a lower-dimensional distribution
  - PPCA allows us to learn a **weight matrix**, which is what allows us to **upsample** the low-dimensional data
  - we assume that our samples are **normally** distributed

- **How does the PPCA model work?**
  - consider data $\underline{x} \in \mathbb{R}^D$
  - say we have a K dimensional **Gaussian** variable:
    $$\underline{\nu} \in \mathcal{N}(\underline{0}, \mathbb{I}_K)$$
  - we then **learn** a matrix $W \in \mathbb{R}^{K \times D}$, such that:
    $$\underline{x} = W\underline{\nu} + \underline{\mu}$$
  - under this assumption:
    $$\mathbb{E}[\underline{x}] = \mu \in \mathbb{R}^D$$
    $$Cov[\underline{x}] = WW^T$$
    since $\underline{x}$ is obtained by applying W to a standard normal ν. Hence, we expect that:
    $$\underline{x} \sim \mathcal{N}(\underline{\mu}, WW^T)$$
  - however, this is a bad model: it generates $\underline{x}$ which lie in a linear **subspace** of dimension K, so this model assigns a likelihood of 0 to any data point outside this subspace (such as general datapoints in $\mathbb{R}^D$)
  - to solve this, we assume the addition of **spherical noise**, such that:
    $$\underline{x} \sim \mathcal{N}(\mu, WW^T + \sigma^2 \mathbb{I})$$
    which allows us to derive data in $\mathbb{R}^D$ by using lower dimensional data

- **How does PCA compare to PPCA?**
  - as $\sigma^2 \to 0$, PPCA explains data in a similar way to standard PCA

3 Question

3.1 Notes Questions

1. **When applying PCA to 0 mean data, why isn't it always possible to get 0 error when $K < D$?**
   - if we have 0 mean, our PCA function becomes:
   $$f(\underline{x}) = VV^T\underline{x}$$
   - to have 0 error, we'd require that $VV^T = \mathbb{I} \in \mathbb{R}^{D\times D}$. However, $K < D$ the rank of $V \in \mathbb{R}^{D\times K}$ is at most K, so $rank(VV^T) \leq K < D$. But $rank(\mathbb{I}) = D$, so it is impossible for $VV^T = \mathbb{I}$.
   - alternatively, $V^T\underline{x}$ will be a point in a K dimensional subspace of D. Hence, $V(V^T\underline{x})$ will keep the data embedded in a K dimensional subspace. Hence, unless the $\underline{x}$ are **already** lying in this subspace, we won't be able to reconstitute all the points back into the full space
   - *Be careful! It is not necessarily the case that we "throw information away" by reducing the dimension. Points in higher dimensional space can be represented by using lower dimensional data (see Hilbert curves, which use lines in 1 dimension to fill a square in 2 dimensions). In principle, PCA could perfectly fit to some finite number of training points, but getting 0 error would be extremly difficult.*
2. **If the covariance of a distribution can be written as $WW^T + \sigma^2\mathbb{I}$, then the cost of evaluating the probability of a datapoint given W, σ^2 becomes $\mathcal{O}(DK^2)$. If we learn W by using SGD, what is the computational cost of the gradient update?**
   - recall, computing gradients by backpropagation requires around the same computational complexity as function evaluation
   - hence, applying backpropagation would also be an $O(DK^2)$ operation
3. **Say we want to fit SVD, by using**

4 Tutorial

1. **Consider applying K-nearest neighbours to the following data:**
   (a) **How would the predictions from regularised linear logistic regression:**
   $$P(y = 1 \mid \underline{x}, \underline{w}, b) = \sigma(\underline{w}^T\underline{x} + b)$$
   and 1-nearest neighbours compare on the dataset?

We can modify the KNN classifier by taking a linear transformation of the data:

$$\underline{z} = A\underline{x}$$

and finding the KNN for the new features $\underline{z}$. We'd like to learn the matrix A, but for $K = 1$, the training error is 0 for almost all A (the nearest neighbour of a training point is itself, which is correctly labelled)
A loss function that could evaluate possible transformations A is the *leave-one-out* LOO classification error, defined as the fraction of errors made on teh training set when the K nearest neighbours for a training item don't include the point being classified.

(b) **Find a matrix A where the 1-nearest neighbour classifier has a lower LOO error than using hte identity matrix for the data above. Explain why your matrix works in about 3 sentences.**

(c) **Explain whether we can fit the LOO error for a KNN classifier by gradient descent on the matrix A.**

(d) **Assume that I have implemented some other classification method where I can evaluate a cost function c and its derivatives with respect to feature input locations: $\bar{Z}$, where Z is an $N \times H$ matrix of inputs. I will use that cod eby creating the feature input locations from a linear transformation of some original features:**

$$Z = XA^T$$

How could I fit the matrix A? If A is an $H \times D$ matrix, with $H < D$, how will the computational cost of this method scale with D?

2. **We now centre our data so it has 0 mean, and fit a linear autoencoder with no bias parameters. The autoencoder is a D-dimensional vector-valued function $\underline{f}$ from D-dimensional inputs $\underline{x}$, using an intermediate K-dimensional "hidden" vector $\underline{h}$:**

$$\underline{h} = W^{(1)}\underline{x}$$
$$\underline{f} = W^{(2)}\underline{h}$$

Assume we want ot find a setting of the parameters that minimises the square error $\|\underline{f}-\underline{x}\|^2$, averaged over training examples.

(a) **What are the sizes of the weight matrices? Why is it usually not possible to get 0 error for $K < D$?**

(b) **It's common to transform a batch of data at one time. Given an $N \times D$ matrix of inputs X, we set:**

$$H = XW^{(1)T} \qquad F = HW^{(2)T}$$

The total square error:

$$E = \sum_{n,d}(F_{nd} - X_{nd})^2$$

has derivatives with respect to the neural network output:

$$\frac{\partial E}{\partial F_{nd}} = 2(F_{nd} - X_{nd})$$

Using the backpropagation rule for matri x multiplication:

$$C = AB^T \implies \bar{A} = \bar{C}B \qquad \bar{B} = \bar{C}^T A$$

write down how to compute derivatives of the cost with respect to $W^{(1)}, W^{(2)}$

(c) **The PCA solution sets:**

$$W^{(1)} = V^T \qquad W^{(2)} = V$$

where the columns of V contain eigenvectors of teh covariance of the inputs. We only really need to fit on ematrix to minimise square error. Tying the weight matrices together:

$$W^{(1)} = U^T \qquad W^{(2)} = U$$

we can fit one matrix U by giving its gradients:

$$\bar{U} = \bar{W}^{(1)T} + \bar{W}^{(2)}$$

to a gradient-based optimiser. Will we fit the same V matrix as PCA?

3. **Some datapoints lie along the one-deimsnsional circumference of a semi-cricle. You could create such a dataset, by:**

$$x_1^{(n)} \sim Uniform[-1, 1]$$
$$x_2^{(n)} = \sqrt{1 - (x_1^{(n)})^2}$$

   (a) **EpExplainxlain why these points can't be perfectly reconstructed when passed through the linear autoencoder in Q3 with $K = 1$.**

   (b) **explain whether the points could be perfectly reconstructed with $K = 1$ by some non-linear decoder: $\underline{f} = \underline{g}(h)$. Here, $\underline{g}$ could be an arbitrary function, perhaps represented by multiple neural network layers. Assume the encoder is still linear: $h = W^{(1)}\underline{x}$.**

   (c) **explain whether the points could be perfectly reconstructed with $K = 1$ by some non-linear encoder: $\underline{h} = \underline{g}(\underline{x})$. Here, $\underline{g}$ could be an arbitrary function, perhaps represented by multiple neural network layers. Assume the encoder is still linear: $\underline{f} = W^{(2)}\underline{h}$.**