ELEC425 Final Exam Notes (without regression stuff)

Definitions and Concepts 5
Machine Learning 5
Deep learning 5
Supervised learning 5
Unsupervised learning 5
Reinforcement learning 5
Feature scaling and normalization 5
Parametric models 5
Non-Parametric models 5
Overfitting 5
Underfitting 6
Cross-Validation 6
N-fold validation 6
Leave-one-out validation 6
Online algorithm 6
Stochastic gradient descent (SGD) 6
Perceptron convergence theorem 6
Impurity 6
Information Gain 6
Bayes Theorem 7
Generative model 7
Discriminative model 7
Maximum likelihood estimation 8
Extended Feature Space 8
Kernel method and trick 8
Gram Matrix 8
Polynomial Order 8
Regularization 9
Latent variables 9
Expectation Maximum (EM) Algorithm 10
Dimensionality Reduction 10
Principal Component Analysis (PCA) and Reconstruction 10
Smoothing 11

ELEC425 Final Exam Notes (without regression stuff)

Classification – Supervised 11
K-Nearest Neighbours (KNN) 11
Training KNN 11
Validate and Test – KNN 11
Implementation Problems – KNN 11
Linear Classifiers 11
Perceptron 11
Perceptron algorithm 12
Decision Tree 12
Learning – Decision Trees 12
Naïve Bayes 13
Training – Naïve Bayes 13
Test – Naïve Bayes 13
Example – Naïve Bayes 13
Naïve Bayes with Discrete Features 13
Learning 13
Gaussian Classifier 14
Learning Gaussian Classifier 14
Obtaining shared covariance matrix 15
Gaussian Naïve Bayes 15
Logistic Regression 15
Training Logistic Regression 15
Logistic Regression vs Perceptron 16
Logistic Regression vs Gaussian Classifier 16
Support Vector Machine (SVM) 16
Clustering 16
K-means 17
K-medians 17
K-Medoids 17
Hierarchical Clustering 17
Mixture of Gaussians 17
Time Series Prediction – Supervised 18
Markov model 18

Learning 18
Hidden Markov Model 19
Viterbi algorithm 19
The forward algorithm 20
Neural Networks Notes 20
Deep Learning 20
Linear Neuron 20
Binary threshold neurons 20
Rectified Linear Neurons (ReLU) 21
Sigmoid Neurons 21
Feed-Forward (FF) Neural Networks 21
How to solve non-linearly separable problems using NN 22
Universal approximator theorem 22
Overfitting in NN 23
Learning by Perturbing Weight 23
Training FF NN 23
1.Forward propagation, compute units and output. 24
2.Compute the loss and errors 24
3.Back propagation, compute derivatives 24
4.Use some existing algorithms to find good parameters 25
Back Propagation 25
self-supervised learning 25
Recurrent Neural Networks (RNN) 25
Memoryless models 26
Limitations of HMM 26
Benefits of RNN 26
Forward propagation of RNN 27
Backward propagation of RNN 27
Difficulty in training RNN 27
Exploding and vanishing gradients 27
Long short-term memory (LSTM) 28
Convolutional Neural Networks (CNN) 29
Convolution stage 29

Convolution vs fully connected 29
Subsampling 30
Max pooling 30
Average pooling 30
Training CNN's: Back propagation 30
How to improve model performance 31
Data augmentation 31
Assembling or averaging models 31
Dropout 31
Autoencoder 31
Regularizing autoencoders 31
Denoising Autoencoders (DAE) 32
Review of pre-needed knowledge 32
Eigenvalues + vectors 32

ELEC425 Final Exam Notes (without regression stuff)

Definitions and Concepts

Machine Learning: give a computer algorithm inputs and or outputs and attempt to learn from it and make a prediction.

Deep learning: set of machine learning algorithms that model high-level abstractions in data by using model architectures (i.e., neural network)

Supervised learning: given example inputs and corresponding desired outputs, predict outputs on future inputs.

Unsupervised learning: given only inputs, automatically discover representations, features, structures, etc.

Reinforcement learning: given sequences of inputs, actions from a fixed set, and scalar rewards/punishments, learn to select action sequences in a way that maximizes expected reward

Feature scaling and normalization: Used to 'regulate' features. If one of the features has a broad range of value, the distance will be dominated by this particular feature if feature normalization is not performed, which is not good.

3 different equations to compute normalization: Min-Max, Mean-Normalization, Standardization

Min-max normalization---it rescales all values of a "feature" (e.g., "taxable income") with:

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

Mean normalization:

$$x' = \frac{x - \text{average}(x)}{\max(x) - \min(x)}$$

Standardization:

$$x' = \frac{x - \bar{x}}{\sigma}$$

Parametric models: parametric models summarize data using a fixed-set of parameters. The number of parameters does not grow. Model-based. (i.e., linear classifiers)

Non-Parametric models: set of parameters is not fixed and often grows in accordance with the dataset's size. instance-based or memory-based models.

Overfitting: a learning algorithm corresponds to closely to a given dataset (memorizes it basically). May fail to fit additional data or predict future observations reliably. Happens with too little data. "overfit to noise". High training error.

ELEC425 Final Exam Notes (without regression stuff)

Underfitting: occurs when a model can't model the training data or generalize new data. Occurs with large datasets. Has both high training and testing error. Can degrade model to a majority classifier (i.e., KNN)

Cross-Validation: A type of validation usually used when your data size is not sufficient for regular validation. This method creates sections of your entire data set, and changes which section is used for validation each time. After each n validations are run the model with the best performance will be chosen.

N-fold validation: break down the data into N folds. Train and test data such that each segment of N gets tested, choose the best performing fold at the end to use.

Leave-one-out validation: used when data is exceptionally scarce. Train on all but 1 example.

Online algorithm: an algorithm that updates parameters at each point.

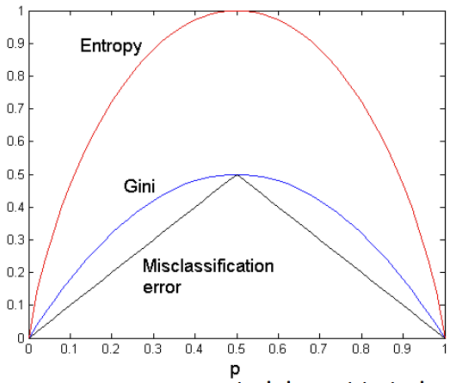
Stochastic gradient descent (SGD): an optimization algorithm typically used in perceptron to minimize the error. SGD is an online algorithm.

Perceptron convergence theorem: if there exists an exact solution (i.e. data is linearly separable) then the perceptron algorithm is guaranteed to find an exact solution in a finite # of steps and will converge. If it is not linearly separable, the perceptron will not converge.

Impurity: Generally refers to the measurement used to build Decision Trees to determine how the features of a dataset should split nodes to form the tree. Few types: GINI, Entropy, Misclassification Errors.

Equations, where j is probability given a class, t is total samples:

- GINI: $GINI(t)$ = 1 – Σ $[p(j\,|\,t)]$^2
- Entropy: $Entropy(t)$ = – Σ $p(j\,|\,t)$ log2 $p\ (j\,|\,t)$
- Misclassification errors: $Error(t)$ = 1 – max_j $P(j\,|\,t)$



Information Gain: a measure of how much information a feature provides about a class. We use information gain when considering splitting data set D at a node based on feature x_i. The split is based on some impurity measures I(D).

ELEC425 Final Exam Notes (without regression stuff)

$$\text{Gain}(D; x_i) = I(D) - \sum_{v \in split(x_i)} \frac{|D_{iv}|}{|D|} I(D_{iv})$$

$$GAIN_{split} = Entropy(p) - \left(\sum_{i=1}^{k} \frac{n_i}{n} Entropy(i) \right)$$

Above is equation for information gain using entropy I(D). The parent node K is split into k partitions where n_i is the number of datapoints in the I'th partition. Reduction in impurity is achieved because of the split.

Bayes Theorem

$$p(\mathcal{C}_k|\mathbf{x}) = \frac{p(\mathbf{x}|\mathcal{C}_k)p(\mathcal{C}_k)}{p(\mathbf{x})}$$

$p(C_k|\mathbf{x})$: posterior distribution
$p(C_k)$: class prior
$p(\mathbf{x}|C_k)$: likelihood, class-conditional distribution
$p(\mathbf{x})$: evidence

Generative model: A model which computes P(Ck | x) using Bayes Theorem during the test stage, and learns P(x | Ck) and P(Ck) during training. This model can also generate new unseen data using the joint distribution P(x | Ck). Note: only need to calculate the numerator portion of the equation.

Discriminative model: These models learn P(Ck | x) directly during training. For example the logistic regression function uses the sigmoid function to learn P(Ck | x) for each data point.

Covariance matrix: based on the density function. The covariance matrix indicates how different features 'Co-vary' with each other.

$$\text{cov}(\mathbf{x}, \mathbf{y}) = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{n}$$

- If cov is positive: both features increase together
- If cov is negative: inverse- when one increases, the other decreases
- If cov is 0: the features are linearly independent.
- Note: Cov(X,X) is var(x). variance equation shown below.

ELEC425 Final Exam Notes (without regression stuff)

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

S^2 = sample variance
x_i = the value of the one observation
$\bar{x}$ = the mean value of all observations
n = the number of observations

Maximum likelihood estimation: used to derive formula for u, σ in univariate Gaussians for closed-form formulae.

Extended Feature Space: the idea that we can map data into a higher-dimensional space which includes some non-linear functions of the original. ϕ(x) is often called the image of x, and the point x can be called the pre-image of ϕ(x).

Kernel method and trick: The kernel trick promotes data into a higher dimensionality by reduce the algorithm down to dot products, and then replacing the dot products with a kernel function, k(x,z). In this higher dimensionality the data should be easier to separate.

Examples of polynomial kernels:

Linear: $\mathbf{x}^T \mathbf{z}$

Polynomial: $(c + g\,\mathbf{x}^T \mathbf{z})^n$

RBF: $\exp(-g\|\mathbf{x} - \mathbf{z}\|^2)$

Sigmoid: $\tanh(c + g\,\mathbf{x}^T \mathbf{z})$

Rules for constructing new kernel functions:

- The sum of two kernels is a kernel
- The product of two kernels is a kernel
- A kernel plus a constant is a kernel
- A scalar times a kernel is a kernel

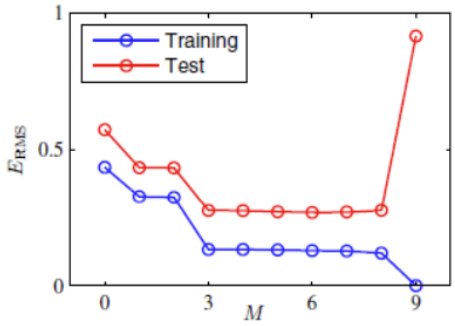
A kernel machine contains 2 modules: the machine learning algorithm itself + kernel function.

To choose which kernel function to use, use a validation set.

Gram Matrix: N-by-N symmetric matrix (N is number of training datapoints) that contains all pairwise dot products between training datapoints. Each element in the matrix is a dot product between the corresponding datapoint xi and xj. Once you have kernelized a ML algorithm and built the gram matrix, you can toss the original data – algorithm will only need to access the gram matrix.

Polynomial Order: Generally referring to its use in regression models. As order increases, models become more complex; training error will decrease, however, test error will increase. Eventually test error goes to 100 and training error goes to 0. (Re: overfitting and underfitting)

ELEC425 Final Exam Notes (without regression stuff)



Regularization: a technique used to control overfitting, add a penalty term to the error function in order to discourage the coefficient w (weight) from reaching large values. Regularization 'smooths' curves.

$$\widetilde{E}(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\{y(x_n, \mathbf{w}) - t_n\}^2 + \frac{\lambda}{2}\|\mathbf{w}\|^2$$

General form:

$$\frac{1}{2}\sum_{n=1}^{N}\{y(x_n, w) - t_n\}^2 + \frac{\lambda}{2}\sum_{j=1}^{D}|wj|^q$$

N = # training points, D is the dimensionality of w.

Few specific types of regularizer: Lasso, Quadratic. Lasso q = 1 (diamond shape) quadratic q = 2 (circle shape). Lasso can give small/sparse models where some weights are near 0.

Regularization aims to balance between minimizing unregularized error and finding small weights.

The regularization terms can be regarded as a constrained term, i.e. minimizing the above error function is same as minimizing:

$$\frac{1}{2}\sum_{n=1}^{N}\{y(x_n, w) - t_n\}^2$$

subject the following constraint:

$$\sum_{j=1}^{D}|wj|^q \leq \eta$$

Lagrange multipliers can be used to convert between the two forms.

Latent variables: In many machine learning problems or models, certain variables may be always unobserved, called latent variables. Latent variables may appear naturally, from the structure of the problem, though sometimes we introduce them on purpose to model complex dependencies.

ELEC425 Final Exam Notes (without regression stuff)

Expectation Maximum (EM) Algorithm: iterative method to find (local) maximum likelihood or maximum a posteriori (MAP) estimates of parameters in statistical models, where the model depends on unobserved latent variables. Typically used in mixture of gaussians and k-means. EM is an optimization strategy for objective functions when we have some latent variables.

- E step: Estimate "expectation" of "missing" or "unobserved" variables from observed data and current parameters.
- M step: Using this "complete" data, find the maximum likelihood parameter estimates.

EM can be used to help classification with some class labels missing in training data.

The EM optimization algorithm guarantees the model's convergence to extrema but does not guarantee finding global extrema.

Dimensionality Reduction: The idea that you can reduce the dimensions of data using an algorithm.

Principal Component Analysis (PCA) and Reconstruction: a type of dimensionality reduction. Two different views:

- "PCA can be defined as the orthogonal projection of the data onto a lower dimensional linear space, known as the principal subspace, such that the variance of the projected data is maximized."
- "Equivalently, it can be defined as the linear projection that minimizes the average projection cost, defined as the mean squared distance between the data points and their projections"

Steps:

1. Given a dataset X = X{x^1…x^(D)} where x^(i) are the values of the ith feature, produce new dataset based on the original by subtracting the means along each feature (mean adjusted data produced)
2. Calculate the covariance matrix
3. Calculate the unit eigenvectors and eigenvalues of the covariance matrix.
4. Order eigenvectors by eigenvalues from highest to lowest.
5. Construct the new 1D space using the first principal component as the basis, or construct a 2D space using (v1,v2) as the basis.
6. Project dataset to 1D space 1 using transformedData = meanAdjustedData * Basis

The larger vector is called the first principal component and subsequent vectors are called second principal component, etc.

Often results in D components. To reduce dimensionality M, ignore D-M components at the bottom of the list.

To get original data from PCA'd data, solve for mean adjusted data. We know that know that (BasisOfNewSpace^T × BasisOfNewSpace)is an identity matrix because the basis consists of unit eigenvectors. MeanAdjustedData = TransformedDataInNewSpace × BasisOfNewSpaceT, OriginalData = MeanAdjustedData + OriginalMean.

If you leave out some eigenvectors, when applying PCA, you'll end up with data loss.

Smoothing: applied in Markov models, leave some probability for unseen cases that were not trained on.

Classification – Supervised

K-Nearest Neighbours (KNN)

KNN is a non-parametric, supervised, discriminative model. KNN uses proximity to make classifications or predictions about the grouping of an individual data point. K neighbours are checked. The optimal K value usually found is the square root of N, where N is the total number of samples. However, looking at an error/accuracy plot can also get you the most optimal K value. If K = 1, its nearest neighbour. Bad K value choices leads to underfitting and overfitting.

Training KNN

Store all the datapoints in some way. Given a test point Xq, find the sphere around Xq containing K points. Classify Xq according to the majority of the K neighbour's class. Choose odd # K to avoid ties.

Validate and Test – KNN

Select your model (whichever K value) and then test on the given set. May require cross-validation to determine which model to use. Good split is 60:20:20 or 70:30 (train, validate, test or train, test)

Implementation Problems – KNN

Learning is cheap, but prediction is expensive – need to retrieve K nearest points from a large dataset for each prediction made. May require k-d trees, locality-sensitive hashing.

Curse of dimensionality: in high dimensions, almost all points are far away. Becomes more expensive to calculate. As such, KNN is good in low-dimensional spaces and datasets where the data isn't too spread.

Linear Classifiers

The goal of linear classifiers is to binarily separate data into two classes by using a hyperplane. Linear classifiers are parametric.

Linear classifiers are governed by the following equation: $y(x) = wTx + w0$, where the vector w controls the decision boundary (perpendicular to the decision boundary) and w0 is the bias. There is d+1 parameters where d is the dimension of a point.

A given point x is at a distance $|y(x)/||w||$ to the decision boundary. The distance from the origin to the decision boundary is $|w0|/||w||$.

If we absorb bias, the equation simply ignores w0.

Perceptron

The perceptron model is basically a single layered neural network and is a linear classifier. The perceptron aims to place a decision boundary to minimize an error function known as perceptron criterion.

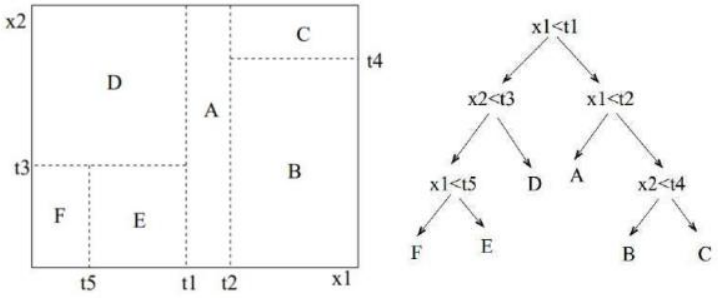
Perceptron algorithm

The perceptron algorithm seeks a weight vector w such that a data point xn in class C1 will have wTϕ(xn) > 0, whereas a data point xn in class C2 have wTϕ(xn) < 0. (+1 for C1, -1 for C2). The perceptron criterion associates a 0 error with any datapoint correctly classified. If incorrectly classified, it tries to minimize the summation of -wTϕ(xn) tn.

```
PerceptronTrain(X, t){ // X: feature matrix; t: true labels (take a value of 1 or -1)
    w = small random values;
    do {
        errors = 0;
        for n =  0 : N-1 {   //loop through all training data points
            x = X (n) // get the feature vector of the n^th data point
            y(x) = f(w^T ϕ(x)) //ϕ(x) can simply just be x (with bias absorbed) or
                               a fixed non-linear transformation of x; f(.) is
                               a nonlinear activation function
                               f(a) = { +1,  a ≥ 0
                                      { -1,  a < 0
            if (y(x) != t(n)){      //η is called learning rate
                w = w + η ϕ(x) t(n);
                errors ++;  }
        }
    } until (errors == 0)
    return w;
```

Decision Tree

Decision trees are a type of parametric model. The idea is to build the trees based on a split of features.



Learning – Decision Trees

Put all examples into the root node

At each node: search all dimensions, choose one that splits data in order to reduce impurity the most

Sort the data cases into the child nodes based on the split

Recurse until a leaf condition:

number of examples at node is too small or

all examples at node have same class, etc.

Prune trees down to some maximum number of leaves.

To test, put the new datapoint into the tree and check the output at the bottom. Decision trees' goal is to minimize the expected sum of impurity at leaves. Finding globally optimal tree is computationally expensive (NP-Hard) and decision trees take a greedy approach (use greedy algorithm).

Decision tree variants: ID3, C4.5,C5.0, CART

Naïve Bayes

Naïve bayes model is a type of generative model. Given an instance, predict the label. To make the prediction, we need to know P(Ck) and P(x|Ck). Remember: we compute P(Ck|x) using Bayes' theorem. Naïve bayes assumes all features are independent and are conditioned on class.

$$p(\mathbf{x}|C_k) = \underbrace{p(x_1, \dots, x_D|C_k) = p(x_1|C_k) \dots p(x_D|C_k)}_{\text{Naive Bayes Assumption}} = \prod_{i=1}^{D} p(x_i|C_k)$$

Training – Naïve Bayes

To get P(x|Ck), we need to first compute P(Xi|Ck). Remember: in the slides, we predicted if someone would play tennis based on the weather conditions (Yes/No)

Test – Naïve Bayes

Given a new instance, we check features based on the training and compute the final probability. We assign x' to the higher of the two.

Example – Naïve Bayes

- Now the goal is to compute

P(Play=*Yes*|**x**') ∝ [P(*Sunny*|Yes)P(*Cool*|*Yes*)P(*High*|Yes)P(*Strong*|*Yes*)]P(Play=*Yes*)
P(Play=*No*|**x**') ∝ [P(*Sunny*|No) P(*Cool*|No)P(*High*|*No*)P(*Strong*|*No*)]P(Play=*No*)

- Remember during training we have learned these:

P(Outlook=*Sunny*|Play=*Yes*) = 2/9	P(Outlook=*Sunny*|Play=*No*) = 3/5
P(Temperature=*Cool*|Play=*Yes*) = 3/9	P(Temperature=*Cool*|Play==*No*) = 1/5
P(Huminity=*High*|Play=*Yes*) = 3/9	P(Huminity=*High*|Play=*No*) = 4/5
P(Wind=*Strong*|Play=*Yes*) = 3/9	P(Wind=*Strong*|Play=*No*) = 3/5
P(Play=*Yes*) = 9/14	P(Play=*No*) = 5/14

- So we can get:

P(*Yes*|**x**') = [P(*Sunny*|Yes)P(*Cool*|*Yes*)P(*High*|Yes)P(*Strong*|*Yes*)]P(Play=*Yes*) = 0.0053
P(*No*|**x**') = [P(*Sunny*|No) P(*Cool*|No)P(*High*|*No*)P(*Strong*|*No*)]P(Play=*No*) = 0.0206

Given the fact P(Play=*Yes*|**x**') < P(Play=*No*|**x**'), we label **x**' to be "Play=*No*".

Naïve Bayes with Discrete Features

Learning

1. Sort data cases into bins according to Ck .
2. Compute marginal probabilities p(Ck) using frequencies.
3. For each class, estimate the distribution of i th variable/feature: $p\ xi\ Ck$
4. Then we can compute posterior as discussed earlier.

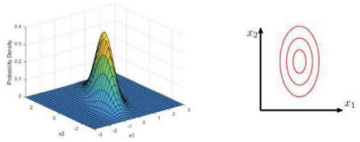
Testing remains the same.

Gaussian Classifier

Gaussian classifiers are a type of generative, class-conditional classifier based on gaussian/normal distributions. Gaussian classifiers are 'closed-form' – no need to use an optimization algorithm. Parameters can be directly obtained using the shared covariance matrix.

Our datapoints often live in a multi-dimensional space.

In a 2-dimensional space, the multivariate Gaussian distribution looks like:



The density function takes the following form:

$$\mathcal{N}(\mathrm{x}|\mu,\Sigma) = \frac{1}{(2\pi)^{D/2}}\frac{1}{|\Sigma|^{1/2}}\exp\left\{-\frac{1}{2}(\mathrm{x}-\mu)^{\mathrm{T}}\Sigma^{-1}(\mathrm{x}-\mu)\right\}$$

where μ is a D-dimensional mean vector, Σ is a D × D **covariance matrix**, and |Σ| denotes the determinant of Σ.

Learning Gaussian Classifier

- Learning class prior p(Ck): easy! just count the training examples like what we did in Naive Bayes.
- Learning the Gaussian p(x|Ck): let's restrict us to the case that is simpler but can show us the key ideas: suppose we have two classes and they share the same covariance matrix Σ.
- Note that the learning phase aims to obtain μ1 , μ2 , and Σ, given training data.

Conclusion: you can use the following formulae to obtain $\boldsymbol{\mu}_1$ and $\boldsymbol{\mu}_2$

$$\mu_1 = \frac{1}{N_1}\sum_{n\in\mathcal{C}_1}\mathrm{x}_n \qquad \mu_2 = \frac{1}{N_2}\sum_{n\in\mathcal{C}_2}\mathrm{x}_n$$

where N_1 is the number of training datapoints in class C_1; N_2 is the number of training datapoints in class C_2. (Note again that $\boldsymbol{\mu}_1$, $\boldsymbol{\mu}_2$ and $\mathbf{x}_n$ are three vectors; each is D-dimensional and D is the number of features)

The above equations are pretty intuitive.

Obtaining shared covariance matrix

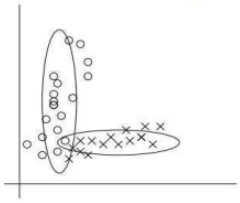
$$\begin{aligned}\boldsymbol{\Sigma} &= \frac{N_1}{N}\mathbf{S}_1 + \frac{N_2}{N}\mathbf{S}_2 \\ \mathbf{S}_1 &= \frac{1}{N_1}\sum_{n\in\mathcal{C}_1}(\mathbf{x}_n - \boldsymbol{\mu}_1)(\mathbf{x}_n - \boldsymbol{\mu}_1)^{\mathrm{T}} \\ \mathbf{S}_2 &= \frac{1}{N_2}\sum_{n\in\mathcal{C}_2}(\mathbf{x}_n - \boldsymbol{\mu}_2)(\mathbf{x}_n - \boldsymbol{\mu}_2)^{\mathrm{T}}\end{aligned}$$

where N is the total number of training datapoints.

Gaussian Naïve Bayes

How if we enforce Naive Bayes assumption on a Gaussian classifier?

- This gives us a Gaussian Naïve Bayes Classifier
- In each class, say C_1, we assume features are independent from each other; remember as we have just discussed, that means the corresponding distribution $p(\mathbf{x}|C_1)$ has a diagonal covariance matrix.
- Training is equivalent to fitting a Gaussian to each class.



- Note that the decision surfaces may be quadratics but not linear, since we do not assume class-conditional Gaussians $p(\mathbf{x}|C_k)$ share the same covariance matrix here.

Logistic Regression

Logistic regressions are a type of discriminative model. They compute the posterior probability of class C1 as P(C1|x) = σ(wT**x**). The logistic function is used here, σ = 1/ (1+exp(-a)) and maps values between 0 and 1. It's derivative is σ(1- σ). Logistic regression models are classification and not regression models.

Training Logistic Regression

In training, logistic regression tries to find **w** to maximize:

$$p(\mathbf{t}|\mathbf{w}) = \prod_{n=1}^{N} y_n^{t_n}\{1-y_n\}^{1-t_n}$$

For the convenience of computation, we can define an error function by taking the negative logarithm of $p(\mathbf{t}|\mathbf{w})$, which results in the form:

$$E(\mathbf{w}) = -\ln p(\mathbf{t}|\mathbf{w}) = -\sum_{n=1}^{N} \{t_n \ln y_n + (1-t_n)\ln(1-y_n)\}$$

To maximize $p(\mathbf{t}|\mathbf{w})$ is same as to minimize E($\mathbf{w}$)

To find the minimum of the error, take the gradient of the error function with respect to $\mathbf{w}$:

$$\nabla E(\mathbf{w}) = \sum_{n=1}^{N} (y_n - t_n)\phi_n$$

where $\phi_n = \phi(x_n)$. Note that same as in Perceptron, you can simply think $\phi(x_n)$ is just x_n, if you do not *preprocess* x_n. ($\phi(x_n)$ means sometimes you want to preprocess x_n, before feeding it into the algorithm; e.g., you can perform some fixed non-linear transformation over x_n)

Once obtaining the derivative, we can use it in an optimization algorithm to obtain w. E.g., we can use the SGD algorithm (the method we used in perceptron), by replacing gradient there with the above gradient, to update w for each training datapoint.

Logistic Regression vs Perceptron

Both are parametric, discriminative. However, logistic regression uses probabilities; perceptron does not.

Logistic Regression vs Gaussian Classifier

Logistic regression is discriminative whereas gaussian classifiers are generative.

For a D-dimensional feature space, logistic regression/perceptron have D+1 adjustable parameters. Gaussian classifiers use D(D+5)/2+1 parameters

- 2D parameters for the means
- D(D + 1)/2 parameters for the (shared) covariance matrix
- 1 parameter for the class prior p(C1)

If you know your data in each class is Gaussian, you should use Gaussian classifiers. Otherwise you can train different types of models and use validation (or cross-validation) to select models.

Support Vector Machine (SVM)

Kernelized maximum-margin hyperplane classifier. For linearly separable data, of all the hyperplanes that separate the data without misclassifying any data points, pick the hyperplane that maximizes the margin. SVM's are sparse – reduce computational load (good for larger datasets).

Clustering

Inputs are continuous or categorical. Goal is to group data cases into a finite number of clusters so that within each cluster all cases have very similar inputs.

ELEC425 Final Exam Notes (without regression stuff)

K-means

For a pre-defined number of clusters (K), randomly initialize a center for each cluster. Alternate between 2 steps:

E: for all datapoints, assign each to the clusters whose centre is closest to this datapoint.

M: update each clusters center to be the mean of all points assigned to that cluster.

The goal of K-means is to find an assignment of data points to clusters and a set of centers such that the sum of the squared Euclidean distances of each data point to its closest cluster centre is a minimum.

K-medians

In the assignment (E) step, assign each point to the cluster whose center is closest to this point.

The M step, take the median along each dimension of features to get center of cluster K.

K-Medoids

Update the new ceter cluster to be one of the data points assigned to that cluster to minimize some predefined distance between datapoints in that cluster and the selected centre (e.g., squared Euclidean)

- Expensive as you have to try every possible point.

Hierarchical Clustering

Break the dataset into a series of nested clusters organized in a tree. 2 types: divisive and agglomerative.

Divisive, start with 1 cluster and divide into sub clusters until each point is in its own cluster.

Agglomerative, start with each In its own cluster and combine clusters until 1 cluster.

Mixture of Gaussians

The idea is to allow a datapoint to belong to multiple clusters at once using a Gaussian probability.

E step: soft assignment, give a data point some probability to belong to different clusters

M step: each centre is moved to the weighted mean of the data, with weights given by soft assignments.

K-means

- E step:

$$r_{nk} = \begin{cases} 1 & \text{if } k = \arg\min_j \|\mathbf{x}_n - \boldsymbol{\mu}_j\|^2 \\ 0 & \text{otherwise.} \end{cases}$$

- M step:

$$\boldsymbol{\mu}_k = \frac{\sum_n r_{nk}\mathbf{x}_n}{\sum_n r_{nk}}$$

Mixture of Gaussians (MOG)

- E step:

$$\gamma(z_{nk}) = \frac{\pi_k \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}_n|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)}$$

- M step:

$$\boldsymbol{\mu}_k = \frac{\sum_n \gamma(z_{nk})\mathbf{x}_n}{\sum_n \gamma(z_{nk})}$$

$$\boldsymbol{\Sigma}_k = \frac{\sum_n \gamma(z_{nk})(\mathbf{x}_n - \boldsymbol{\mu}_k)(\mathbf{x}_n - \boldsymbol{\mu}_k)^{\mathrm{T}}}{\sum_n \gamma(z_{nk})}$$

$$\pi_k = \frac{\sum_n \gamma(z_{nk})}{N}$$

20

Time Series Prediction – Supervised

Markov model

Markov models are used in time-series prediction, supervised learning. Markov models are chain-structured processes where future states depend only on the current state, not the sequence of events that preceded it.

first order considers the single previous state, second order considers 2 previous states, etc.

first order: P(x) * P(xt | xt-1)

second order: P(x) *P(xt | xt-1, xt-2), in other words: P(x) * P(x2|x1)*P(x3|x2,x1)

Learning

Count and calculate the transition probabilities based on the training data

Hidden Markov Model

A large number of sequences are observed but there are hidden states that generate the individual sequence

Decoding

GIVEN an HMM specified by $\boldsymbol{\theta}$, and a sequence $\mathbf{x}$,
FIND the sequence $\mathbf{z}$ that maximizes $p(\mathbf{z}|\mathbf{x},\boldsymbol{\theta})$,
i.e., $\mathbf{z}^* = \text{argmax}_{\mathbf{z}}\, p(\mathbf{z}|\mathbf{x},\boldsymbol{\theta})$

Evaluation

GIVEN an HMM specified by $\boldsymbol{\theta}$, and a sequence $\mathbf{x}$,
FIND $p(\mathbf{x}|\boldsymbol{\theta}) = \sum_{\mathbf{z}}\, p(\mathbf{x}, \mathbf{z}|\boldsymbol{\theta})$

Learning

GIVEN a set of observed sequences $\mathbf{X}$
FIND parameters $\boldsymbol{\theta}$ that maximizes $p(\mathbf{X}|\boldsymbol{\theta})$,
i.e., $\boldsymbol{\theta}^* = \text{argmax}_{\boldsymbol{\theta}} p(\mathbf{X}|\boldsymbol{\theta})$

The problem: find $\mathbf{z}^*$:

$\mathbf{z}^* = \text{argmax}_{\mathbf{z}}\, p(\mathbf{z}|\mathbf{x},\boldsymbol{\theta}) = \text{argmax}_{\mathbf{z}}\, (p(\mathbf{x},\mathbf{z}|\boldsymbol{\theta}) / p(\mathbf{x},\boldsymbol{\theta}))$

$= \text{argmax}_{\mathbf{z}}\, p(\mathbf{x},\mathbf{z}|\boldsymbol{\theta})$

We have already known how to calculate $p(\mathbf{x},\mathbf{z}|\boldsymbol{\theta})$.

The naïve method of finding z* is to compute all $p(\mathbf{x},\mathbf{z}|\boldsymbol{\theta})$ for all values that $\mathbf{z}$ can takes.

Q: What's the problem of this naïve method?

A: Exponential numbers of **z**

Viterbi algorithm

Algorithm in HMM for obtaining the maximum a posteriori probability estimate of the most likely sequence of hidden states. Using for solving decoding issue.

Initialization:
rows = # hidden states
columns = n

First (n = 1) column:
start prob. * emission prob. of first observation

Consecutive (n > 1) column:
emission prob. of nth observation max((n-1)th column transition prob.)

Problems: underflow of probabiltiies, numbers become very small. Mitigated by using logs

ELEC425 Final Exam Notes (without regression stuff)

The forward algorithm

used to calculate a 'belief state': the probability of a state at a certain time, given the history of evidence.

Input: $\mathbf{x} = x_1 \ldots\ldots x_n$, and the model $\boldsymbol{\theta} = (\boldsymbol{\pi}, \mathbf{A}, \mathbf{E})$

Initialization (k = 1 to K):

$F_k(1) = (e_{k,x_1})\ \pi_k$ (note that $\{\pi_k\}$ are the initial probabilities)

Iteration (for t = 1 to n-1, and for k' = 1 to K):

$F_{k'}(t+1) = (e_{k',x_{t+1}})\ \Sigma_k\ (a_{k,k'}\ F_k(t))$, where k = 1 to K.

Termination:

$p(\mathbf{x}) = \Sigma_z\ p(\mathbf{x}, \mathbf{z}) = \Sigma_k\ F_k(n)$

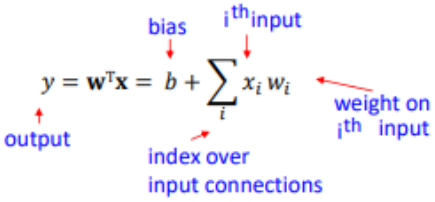
To train the HMM, use EM algorithm estimating θ if we know the sequences of hidden states.

Neural Networks Notes

Deep Learning: A set of machine learning algorithms that model high-level abstractions in data by using model architectures that are known as neural networks).

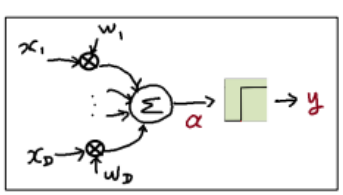
Linear Neuron

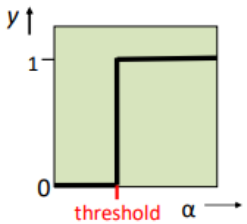
A linear neuron computes the weighted sum of input features and outputs as a linear function. Multi-layered sets of linear neurons still produce a linear model.



Binary threshold neurons

Use a threshold to determine if output is 1 or 0. Basically check if calculated value is greater than a threshold.





$\alpha = b + \sum_i x_i w_i$

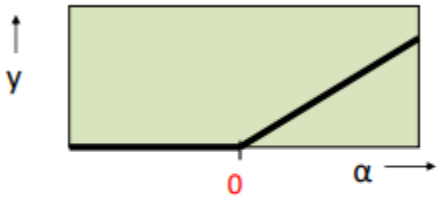
$$y = \begin{cases} 1 \text{ if } \alpha > \text{threshold} \\ 0 \text{ otherwise} \end{cases}$$

Rectified Linear Neurons (ReLU)

Compute a weighted sum of the inputs.

$$\alpha = b + \sum_i x_i w_i$$

$$y = \begin{cases} \alpha \text{ if } \alpha >= 0 \\ 0 \text{ otherwise} \end{cases}$$



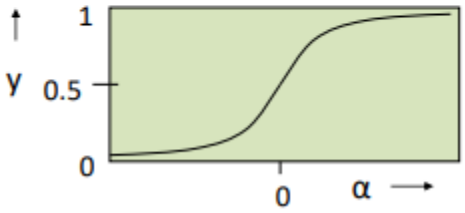
i.e., $y = \max(0, \alpha)$

Sigmoid Neurons

Real-valued output and bounded function of their total imput. Typically they use a logistic function. Very easy derivative which makes learning easy.

$$\alpha = b + \sum_i x_i w_i$$

$$y = \frac{1}{1 + e^{-a}}$$

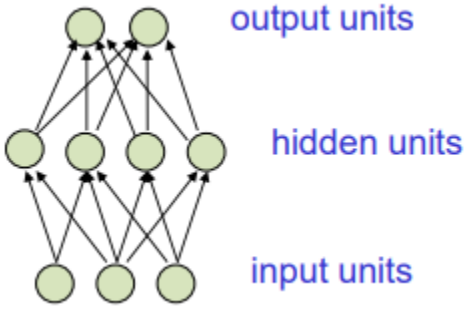


Feed-Forward (FF) Neural Networks

Most basic type of neural network.

The first layer is the input and last layer is output. If there is more than 1 hidden layer, it is called a neural network. They compute a mapping from input to output. The activities of the neurons in each layer are a non-linear function of the activities in the layer below.

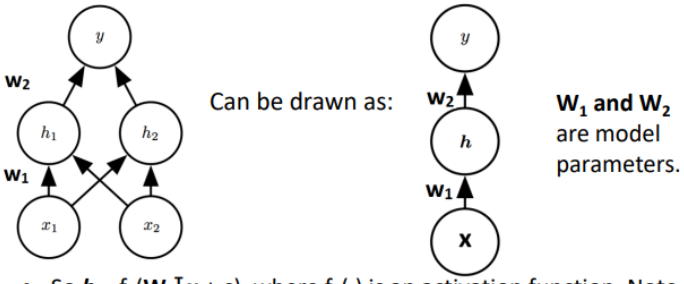
FF networks are fully connected between 2 adjacent layers.



How to solve non-linearly separable problems using NN

We will use a network with 1 hidden layer.

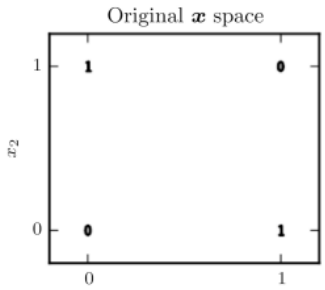
To solve XOR with a simple neural network, look at the lower layer first.



- So $\boldsymbol{h} = f_1(\mathbf{W}_1^T \mathbf{x} + \mathbf{c})$, where $f_1(.)$ is an activation function. Note that $\mathbf{W}_1$ is a matrix.
- $\boldsymbol{y} = f_2(\mathbf{W}_2^T \boldsymbol{h} + \mathbf{b})$, where $f_2(.)$ is also an activation function. Note that **b** and **c** are biases. To help explain the model, we explicitly list biases but they can be absorbed into **W**, as we have discussed earlier in the course.

XOR is a classic example that is not linearly separable:

$f(x_1=1, x_2=1) = 0$
$f(x_1=0, x_2=0) = 0$
$f(x_1=1, x_2=0) = 1$
$f(x_1=0, x_2=1) = 1$



Let's look at the lower layer first:

$$\boldsymbol{h} = \max(0, \mathbf{W}_1^T \mathbf{x} + \mathbf{c})$$

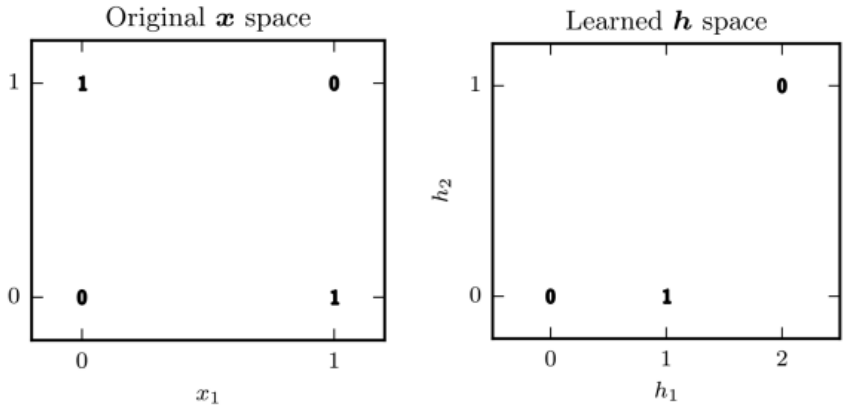
Suppose we have already learn the model:

$$\mathbf{W}_1 = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$$

$$\mathbf{c} = \begin{pmatrix} 0 \\ -1 \end{pmatrix}$$

We see that the lower layer NN maps data (left side) that are not linearly separable in the original space to be linearly separable (right).

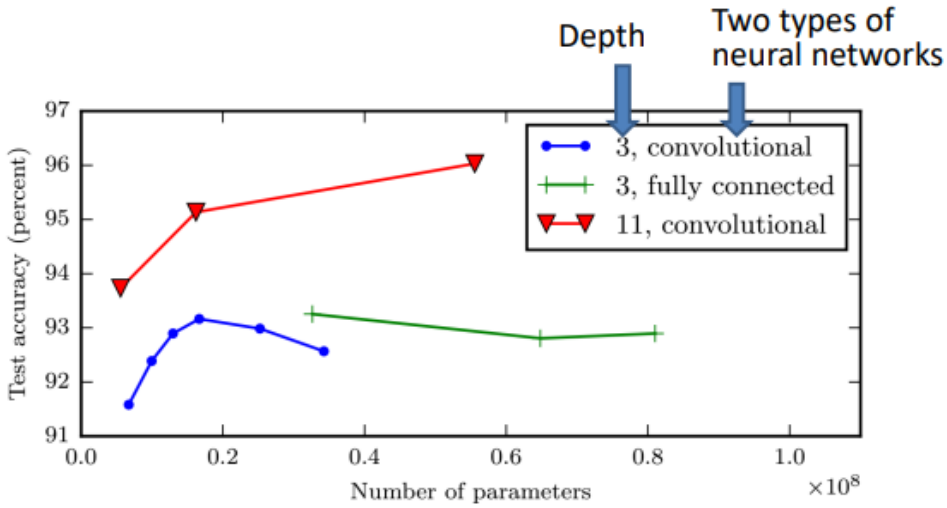
The upper layer neural net can classify the data accordingly now.

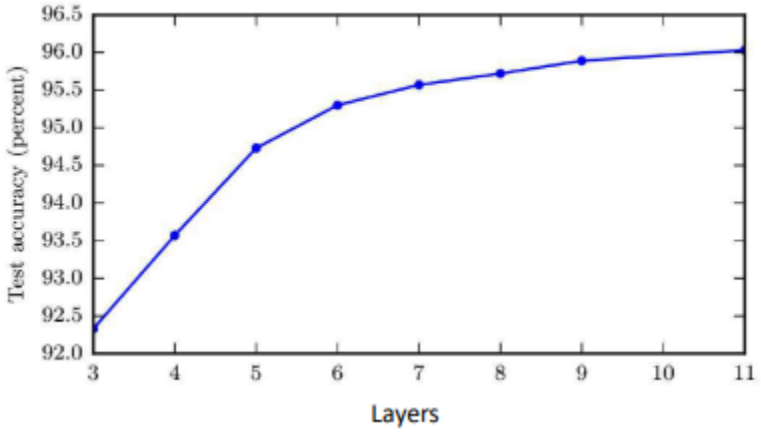


Universal approximator theorem

Theorem that networks with one hidden layer is enough to represent an approximation of any function to an arbitrary degree of accuracy.

Overfitting in NN





Learning by Perturbing Weight

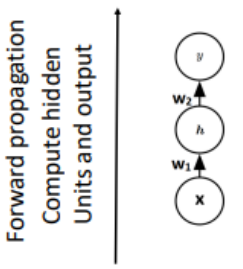
TL;DR bad, this is where you randomly perturb some/all parameters to see if performance increases.

Training FF NN

Key idea is to minimize an error function by computing the derivative with respect to model parameters and set them to zero. Once you get the derivatives, you can choose whatever optimization algorithm you want. Steps:

ELEC425 Final Exam Notes (without regression stuff)

1.Forward propagation, compute units and output.



For the left example, forward propagation computes:

- $\boldsymbol{h} = f_1(\mathbf{W_1}^T\mathbf{x} + c)$
- $\boldsymbol{y} = f_2(\mathbf{W_2}^T\boldsymbol{h} + b)$

For deeper networks, just do the same computation in the same bottom-up way.

2.Compute the loss and errors

First compute the discrepancy between each output y_j and its target value t_j (ground truth).

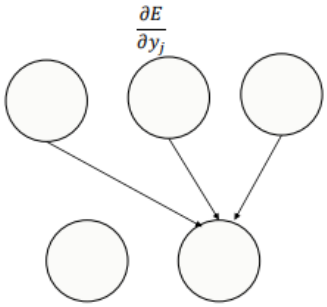
- There are many error functions to use.
- Suppose we use squared error here.

$$E = \sum_j \frac{1}{2}(y_j - t_j)^2$$

$$\frac{\partial E}{\partial y_j} = y_j - t_j$$

Compute the derivative of errors w.r.t. each output y_j.

Then we will use back propagation to compute the derivative of errors w.r.t. parameters at lower layers of the network.



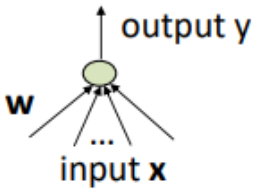
3.Back propagation, compute derivatives

Back propagation applies the derivative chain rule to the network. Work top→down.

For example, consider one neuron where the activation function is a logistic function.

We know the derivative is

y(1-y). using a y calculated in forward propagation, it is very easy.



$$y = \frac{1}{1 + e^{-a}}$$

$$\alpha = b + \sum_i x_i w_i$$

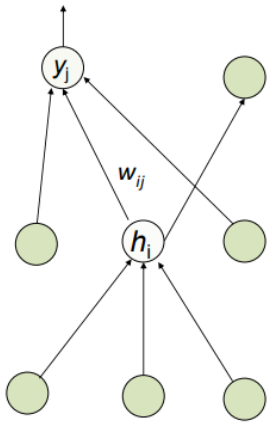
The total derivative of the weighted sum,

$$\alpha = b + \sum_i x_i w_i$$

$$\frac{\delta \alpha}{\delta w_i} = x_i$$

$$\frac{\delta \alpha}{\delta x_i} = w_i$$

The Error Derivatives at the Top Two Layers



$$\frac{\partial E}{\partial \alpha_j} = \frac{\partial E}{\partial y_j}\frac{dy_j}{d\alpha_j} = \frac{\partial E}{\partial y_j} y_j(1-y_j)$$

Again, y_j has been computed in forward propagation.

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial \alpha_j}\frac{\partial \alpha_j}{\partial w_{ij}} = \frac{\partial E}{\partial \alpha_j} h_i$$

Also, h_j has been computed in forward propagation.

$$\frac{\partial E}{\partial h_i} = \sum_j \frac{\partial E}{\partial \alpha_j}\frac{d\alpha_j}{dh_i} = \sum_j \frac{\partial E}{\partial \alpha_j} w_{ij}$$

We keep performing this top-down on the network.

To check if backprop was correct, look at definition of derivative.

If the backprop is implemented correctly, we should have

$$\frac{dE}{d\theta} \approx \frac{E(\theta+\epsilon)-E(\theta-\epsilon)}{2\epsilon} \qquad (1)$$

In neural network, **θ** is a vector that includes the weights **w** of all layers (including biases). The outline of the checking algorithm is:

- We randomly pick a **θ**, and then compute $E(\boldsymbol{\theta}+\epsilon)$ and $E(\boldsymbol{\theta}-\epsilon)$ with forward propagation
- We also compute $E(\theta)$ with forward propagation and based on that compute $\frac{dE}{d\boldsymbol{\theta}}$ with backprob.
- Check if equation (1) above is satisfied or not.

4.Use some existing algorithms to find good parameters

Back Propagation

Back propagation is the algorithm used to compute derivatives in feed-forward neural networks.

self-supervised learning: blurs the distinction between supervised and unsupervised learning. uses methods designed for supervised learning, but doesn't require a separate training signal

Recurrent Neural Networks (RNN)

RNN's are used for sequential data or time series data. RNN has one layer of hidden units to remember history information, so the depth of the network along the timeline corresponds to the number of timesteps.

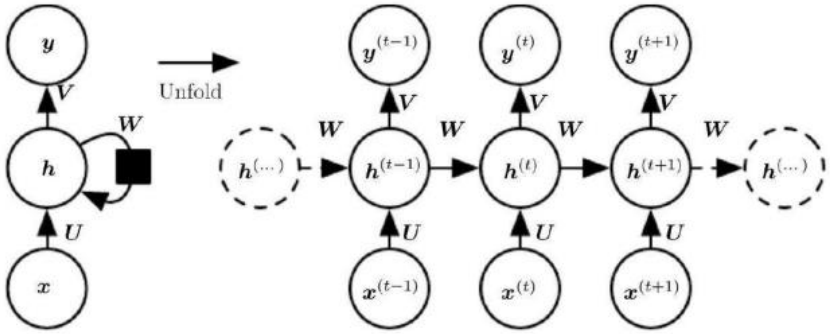
Two cases: turn an input sequence into an output sequence or teach a model by training it to predict the next term in the input sequence.

ELEC425 Final Exam Notes (without regression stuff)

RNN is more powerful due to 3 properties:

1. Hidden-layer units (neurons) save continuous values (rather than one-of-K states in HMM) and allow RNN to store a lot more information about the history.
2. Neurons are updated with non-linear functions instead of being controlled by a simple transition matrix.
3. The output layer at each time step contains non-linear neurons, which are not controlled by a simple emission matrix as in HMM.

RNN can compute anything that can be computed by your computer (Turing complete).



Above is the 2 representations of a RNN. Both map an input sequence x to a corresponding sequence output y. both describe that h is updated when an input is taken in. at each step, an erorr/loss can be used to measure how far each y is from the corresponding training target/true label.

Memoryless models

This type of model has no memory about a long history. (i.e. normal Markov model)

Example 2: FFNN can be designed to predict next term in the input sequence. This generalized the above Markov models by using one or more layers of non-linear hidden units.

Limitations of HMM

HMM have a discrete 1-of-K hidden state and transitions between states are controlled by a transition matrix. At any time instance, all the hidden states need to be selected from. When there are lots of hidden states (latent variables) HMM requires large data bandwidth. 100 bits in the first half, need 2^100 in the second half. can only remember log(K) bits of history.

So basically they need to have all states saved somewhere / be able to chosen from and HMM is memory-heavy.

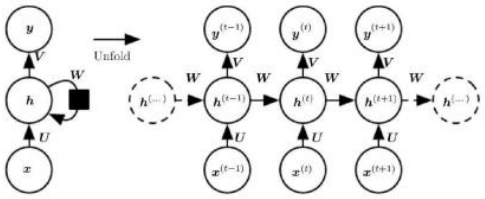
Benefits of RNN

Distributed hidden states allows for more efficient storage of past information.

- Basically more powerful, better memory manipulation, can update hidden states better.

Forward propagation of RNN

Again, the forward propagation of RNNs is simple:



$$\begin{aligned} \boldsymbol{a}^{(t)} &= \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)} \\ \boldsymbol{h}^{(t)} &= \tanh(\boldsymbol{a}^{(t)}) \\ \boldsymbol{o}^{(t)} &= \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)} \\ \boldsymbol{y}^{(t)} &= \text{softmax}(\boldsymbol{o}^{(t)}) \end{aligned}$$

– You can see that layer **h** uses a *tanh* activation here but you can use other activation function as well. Note that **W**, **U** and **V** are shared at all different time steps. 13

Backward propagation of RNN

Similar to FF but RNN reuses the same weights at different timesteps, so compute gradients as usual then modify the gradients so that they satisfy the constraints.

we compute: $\frac{\partial E}{\partial \mathbf{W}^{(t+1)}}, \frac{\partial E}{\partial \mathbf{W}^{(t)}}, \dots$

use $\frac{\partial E}{\partial \mathbf{W}^{(t+1)}} + \frac{\partial E}{\partial \mathbf{W}^{(t)}} + \cdots$ as $\Delta \mathbf{W}$ to update $\mathbf{W}$

So if the weights started off satisfying the constraints, they will continue to satisfy them.

Difficulty in training RNN

RNN is more powerful than HMM, but difficult because:

1. Training time = consuming and less tractable when computers were slow

2. difficulty also due to vanishing / exploding gradient

Exploding and vanishing gradients

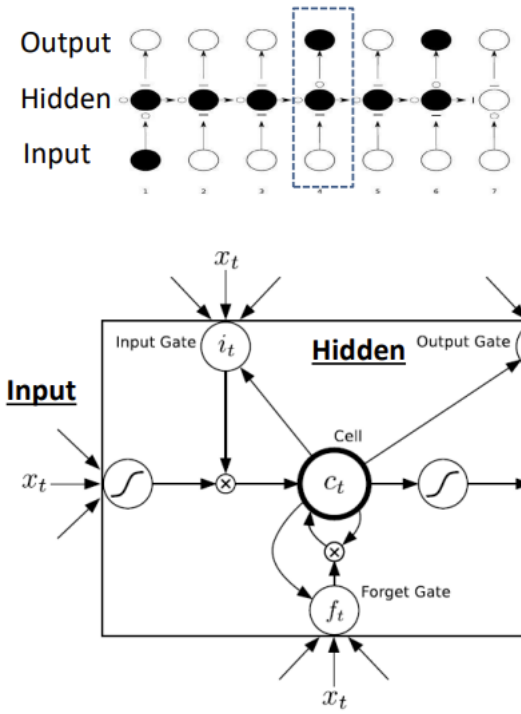
The change in magnitude of the gradients as back propagation occurs.

- If the weights are small, gradients shrink exponentially, likewise for big weights (grow exponentially)
- Activation function can also cause vanishing gradients.

Few hidden layers, not a huge issue; but RNN is often trained on long sequences, so gradients easily explode / vanish here. As such, RNN has difficulty dealing with long-range dependencies. To deal with, use LSTM or Gated Recurrent Unit (GRU).

Long short-term memory (LSTM)

An architectural-solution to RNN's problems. Use gates to control information flow in RNN. By setting gates to open or close, can allow the information to pass in a long distance. LSTM learns soft gates with values 0/1.



- The lower figure (a LSTM block) shows the details of the upper figure in a time step t.
- In each step, we have a vector of memory cells $\mathbf{c}_t$, a vector of output units $\mathbf{y}_t$, and vectors of input, output, and forget gates $\mathbf{i}_t$, $\mathbf{o}_t$, and $\mathbf{f}_t$.
- The input gate $\mathbf{i}_t$ decides how much information is allowed to get in memory cell from the input, and the forget gate $\mathbf{f}_t$ (should actually be called memory gate) decides how much information is allowed from the previous memory. 10

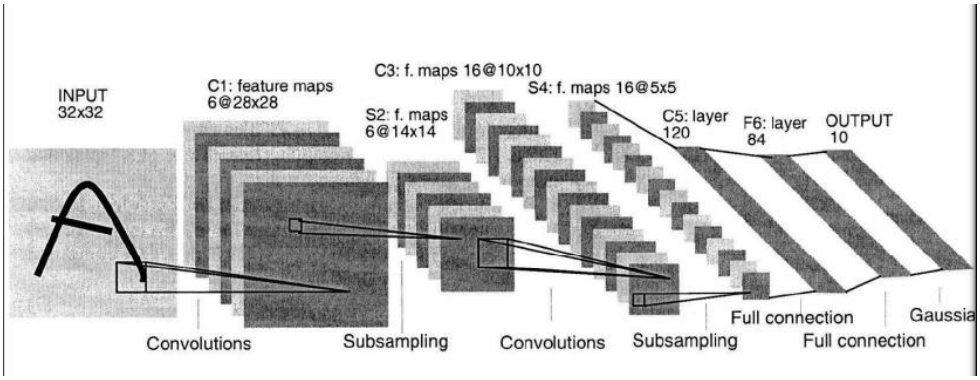
A bar represents gate closed. Open circle represents gate open. The output is a function of the output gate multiplied by the tanh(cell)→ $o_t \tanh(c_t)$

Information gets into the cell whenever its input gate is on. Information stays in the cell so long as its forget gate is on. Information can be read from the cell by turning on its output gate.

LSTM can be extended to recursive LSTM and function as a tree structure.

Convolutional Neural Networks (CNN)

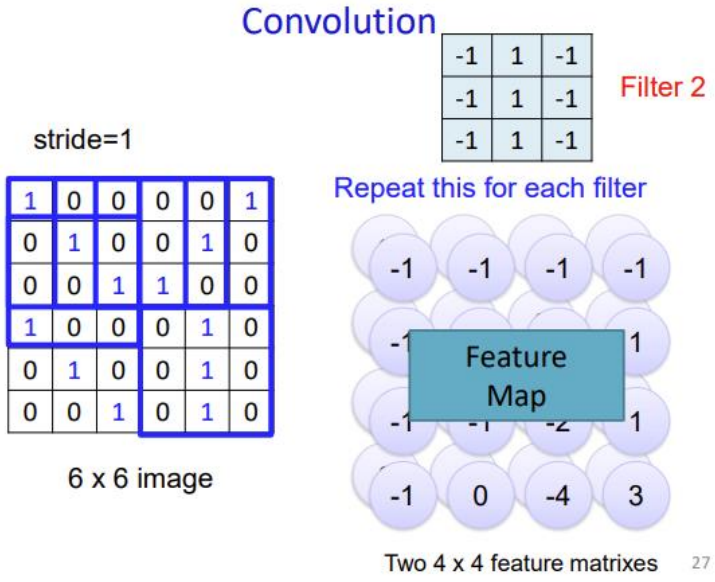
- Uses filters/kernels of replicated units to share parameters and uses a subsampling operation. Idea is that we don't need all edges to be fully connected in the network.



Some patterns are much smaller than a whole image and certain patterns can appear in different positions of images.

Convolution stage

The first key component in CNN, the filter shifts on input to detect local features. At the convolutional layer of a CNN, we often use multiple filters. Filters are often a 3x3 matrix. We take the dot product, and if the sum of dot products is what we want, then the subset of the image has whatever the filter was looking for. We create feature maps after applying the filter onto the original image with whatever given stride.



Convolution vs fully connected

A fully connected layer refers to a neural network in which each input node is connected to each output node. In a convolutional layer, not all nodes are connected. Convoluted has fewer parameters as a result.

Subsampling

We can subsample the pixels to make the image smaller, so fewer parameters are used to characterize the image. Less computation, view of each filter at an upper layer can see a wide range. Subsampling reduces the size of a map to reduce complexity.

Max pooling

Break the map into pools and recreate the map by taking the largest value from each pool. It introduces invariance in the maps.

Average pooling

Break the map into pools and recreate the map by using an average of a pol for reconstruction.

How CNN controls number of model paramets

1. Reduce number of connections
2. Shared weights on edges
3. Pooling reduces complexity.

Training CNN's: Back propagation

Backpropagation for the subsampling layer:

- Max pooling: If you use max pooling, for a pooling area with N-by-N input units, you only need to backprop gradient to the unit that has the max value.
    - If you make a small change in another non-max (i.e., nonwinning) unit, it will not change the pooling results; i.e., gradient is zero w.r.t. that non-winning unit.
- Average pooling: If you use average pooling, it is simple. You just take derivative w.r.t. the corresponding units (i.e., the error is multiplied by 1/N2 and assigned to the whole pooling area (all units get this same gradient value)).

How can we calculat the gradient w.r.t. the weights of a filter?

Remember when a filter is applied to different area **a**, **b, c,** …, the weights ($\mathbf{W}^{(a)}$, $\mathbf{W}^{(b)}$ and $\mathbf{W}^{(c)}$ are shared.

– Similar to the discussion in RNNs, we compute the gradients as usual, and then modify the gradients so that they satisfy the equivalent constraint. That is:

$$\text{we compute:} \quad \frac{\partial E}{\partial \mathbf{W}^{(a)}}, \frac{\partial E}{\partial \mathbf{W}^{(b)}}, \frac{\partial E}{\partial \mathbf{W}^{(c)}} \dots$$

$$\text{use} \quad \frac{\partial E}{\partial \mathbf{W}^{(a)}} + \frac{\partial E}{\partial \mathbf{W}^{(b)}} + \frac{\partial E}{\partial \mathbf{W}^{(c)}} \dots \text{ as the gradient}$$

– So the weights will satisfy the equivalent constraint.

ELEC425 Final Exam Notes (without regression stuff)

How to improve model performance

Data augmentation

Automatically augment existing training data to create larger training data. Data augmentation helps achieve better performance on things like image classification. It is less intitive to use data augmentation in some other harder AI tasks such as NLP. An example of data augmentation is applying rotations, scaling, translation on an image.

Assembling or averaging models

Averaging/assembling the models can lead to better performance.

- Same training data, different models. Run each test datapoint and models, let them vote.
- Combine models by averaging their output probabilities

Dropout

During training, for each input datapoint, randomly ignore each hidden unit with a certain probability. Works well on neural network to improve performance.

Dropout can be seen as sampling from different 'thinned' architectures. Dropout can also be used to take an average of hidden units, serving a as a regularizer to deal with overfitting. You can try and overfit data then regularize it to avoid underfitting.

Dropout works better than lasso and quadratic regularizer for NN.

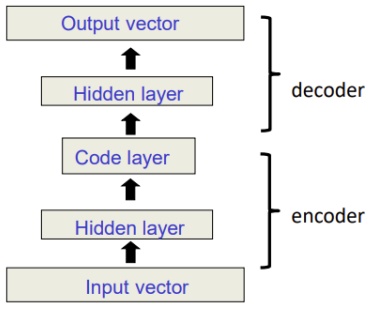
Autoencoder

Type of unsupervised neural network. Autoencoders are trained to copy its input to its output. We hope that the network attempts to remember the most salient information in order to reconstruct the data with fewest errors.

The input-to-hidden part is called an encoder.

The hidden-to-output part corresponds to a decoder.

Autoencoders can be used for dimensionality reduction as well as feature learning.



Autoencoders help reduce noise in data. the process of compressing input data, encoding it, and then reconstructing it as an output, autoencoders allow you to reduce dimensionality and focus only on areas of real value.

Regularizing autoencoders

If the encoder/decoder are too capable, it isn't necessarily a good thing. If they are too capable, the autoencoder can perform copying without extracting salient information about the data into the code layer. to solve, regularize the network to restrict the complexity/capability of encoder/decoder.

Denoising Autoencoders (DAE)

A denoising autoencoder (DAE) first corrupts input data (e.g., by adding background noise). It is then trained to predict the original, uncorrupted data point as its output. These types of encoders perform better than standard autoencoders.

Procedure:

1. Choose a training sample from the training data
2. Obtain corrupted version from corruption process
3. Use training sample pair to estimate reconstruction

Review of pre-needed knowledge

Eigenvalues + vectors

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

If a matrix ***A,*** vector ***v,*** and scalar λ satisfy the above equation, then

- **v** is an *eigenvector* of ***A***. (In this class we always scale the length of an eigenvector to be the unit length 1.)
- λ is called the *eigenvalue* associated with **v**

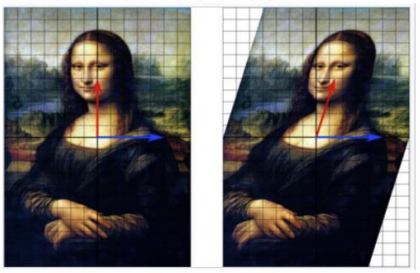
$$\begin{pmatrix} 1 & 0.3 \\ 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} = 1 \times \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

eigenvalue eigenvector

For any vector **v**, we can regard ***Av*** as applying a linear transformation ***A*** upon the vector **v**.

$$\begin{pmatrix} 1 & 0.3 \\ 0 & 1 \end{pmatrix} \times \begin{pmatrix} 0 \\ 5 \end{pmatrix} = \begin{pmatrix} 1.5 \\ 5 \end{pmatrix}$$ (See the red vector)

$$\begin{pmatrix} 1 & 0.3 \\ 0 & 1 \end{pmatrix} \times \begin{pmatrix} 5 \\ 0 \end{pmatrix} = \begin{pmatrix} 5 \\ 0 \end{pmatrix}$$ (See the blue vector)



12

For any eigenvector **v** of **A** and scalar *a*:

$$\mathbf{A}a\mathbf{v} = \lambda a\mathbf{v}$$

so you can always choose eigenvectors to have length of 1:

$$\sqrt{{v_1}^2+\ldots+{v_n}^2} = 1$$

where **v** = (v_1, v_2, …, v_n).

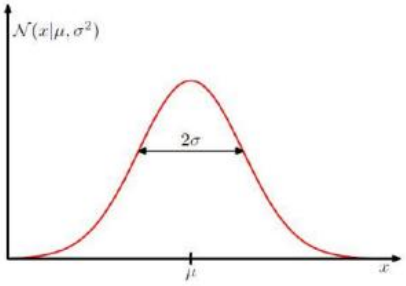
Note that most machine learning or statistics tools will give you eigenvectors of length 1, when you use them to obtain eigenvectors for a matrix.

If ***A*** has multiple eigenvectors and if they are orthogonal to one another, they can be used as a new basis for a n-dimensional vector space.

Univariate Gaussian

Remember that in the case of single-variable Gaussian (i.e., univariate Gaussian), the density function can be written as

$$\mathcal{N}(x|\mu,\sigma^2) = \frac{1}{(2\pi\sigma^2)^{1/2}}\exp\left\{-\frac{1}{2\sigma^2}(x-\mu)^2\right\}$$



If we can learn the class-conditional density $p(\mathbf{x}|C_k)$ through training, we are able to predict the label for a datapoint **x** (draw a simple example).

$$p(\mathcal{C}_k|\mathbf{x}) = \frac{p(\mathbf{x}|\mathcal{C}_k)p(\mathcal{C}_k)}{p(\mathbf{x})}$$