Neural Networks: A Concise Overview

Contents

I. Introduction

- Computer science, artificial intelligence, machine learning & neural networks:
  - *computer science*, or CS, is the study of the principles and use of computers
  - *artificial intelligence*, or AI, is a subfield of CS that studies tasks that normally require human intelligence
  - *machine learning*, or ML, is a subfield of AI that evolved from the study of pattern recognition and computational learning theory
  - *neural networks*, or NN, ia a subfield ML that is inspired by biological neural networks

$$\text{CS} \longrightarrow \text{AI} \longrightarrow \text{ML} \longrightarrow \text{NN} \tag{1}$$

- Machine learning:
  - ML is the science of getting computers to act without being explicitly programmed
  - ML is applied on problems where
    1) data is available
    2) a pattern or correlations exist
    3) mathematical formulation is not practical or possible
- Learning:
  - in nature, cause precedes effect
    $$\text{cause} \to \text{effect} \tag{2}$$
    in learning, observations lead to constructing cause
    $$\text{effect} \to \text{cause} \tag{3}$$
  - formalization of learning $(\mathcal{H}, \mathtt{A})$:
    * $\mathcal{H} = \{h_i\}$ is a hypothesis set,
    * a learning algorithm, $\mathtt{A}$, picks the *final hypothesis* g
    $$\mathcal{H} = \{h_i\} \xrightarrow{A} g \in \mathcal{H} \tag{4}$$
  - learning versus memorization:
    * learning allows for generalization, whereas memorization does not, see Section III
- Paradigms of learning:
  1) supervised learning, Sections IV and V
     - supervised learning implies data is available for training with the desired outcome
     - in classification, the correct outcome is referred to as the *label*
     - at present, supervised learning is the prevalent learning paradigm
  2) unsupervised learning, Section VI
     - no labels available during training
  3) reinforcement learning,
     - no supervision, only a reward system
     - feedback is delayed, not instantaneous
     - for applications where time matters, i.e. sequential, not i.i.d.
     - e.g. backgammon
- Neural networks:
  - a neural network is sometimes referred to as *artificial neural network* or ANN
  - NN is hands-on, empirical field for now
    * experimentation over deep thoughts
  - a NN is composed of neurons, see Section II-B
  - little is known about biological learning to provide guidance for learning of NNs
  - NNs come in two flavors:
    * feedforward, see Section IV, and
    * recurrent, see Section V
  - NNs have been successful in recent years and are the main focus of this write-up [1]
- Factors of variation:
  - when analyzing a speech, the *factors of variation* include the speakers age, their sex, their accent and the words that they are speaking

  - when analyzing an image of a car, the factors of variation include the position of the car, its color, and the angle and brightness of the sun
  - unfortunately, many of the factors of variation influence every single piece of data we are able to observe
  - it is usually needed to disentangle the factors of variation and discard the ones that we do not care about
- Features in a representation:
  - the performance of a machine learning algorithm depends on the *representation* of the data they are given
  - each piece of information included in the representation is known as a *feature*
  - feature extraction from input data is nonlinear
  - features reduce the input data dimensionality by trying to extract useful information
- Template matching:
  - template matching is a simple feature extractor, that may work if factors of variations are limited
- Representation learning:
  - externally identifying relevant features is difficult
  - current philosophy is to use NN to discover not only the mapping from representation to output but also the representation itself
  - this approach is known as *representation learning*

A. Probabilistic formulation

- Interpreting NN as generating a conditional probability: target distribution
  - consider a NN with supervised learning, where the data & label pairs $\{(\boldsymbol{x}_n, \boldsymbol{y}_n)\}$, are available
    - $\boldsymbol{x}_n$ is a d dimensional random vector, representing the input
    - $\boldsymbol{y}_n$ is a d_o dimensional vector, representing the observed output,
  - a NN models the conditional probability of the map
$$\boldsymbol{x}_n \xrightarrow{P(\boldsymbol{y}|\boldsymbol{x})} \boldsymbol{y}_n \tag{5}$$
    where $P(\boldsymbol{y}|\boldsymbol{x})$ is a conditional probability function
  - $P(\boldsymbol{x})$ & $P(\boldsymbol{y}|\boldsymbol{x})$ are in general unknown
  - $(\boldsymbol{x}_n, \boldsymbol{y}_n)$ can be generated from $P(\boldsymbol{x}, \boldsymbol{y})$, where
$$P(\boldsymbol{x}, \boldsymbol{y}) = P(\boldsymbol{x})\, P(\boldsymbol{y}|\boldsymbol{x}) \tag{6}$$
  - $P(\boldsymbol{y}|\boldsymbol{x})$, is referred to as the *target distribution*
    - the target distribution is the quantity that needs to be learned
- Equivalent formulation: target function
  - the target distribution can be interpreted as a deterministic mapping, known as the *target function*
$$f(\boldsymbol{x}_n) \triangleq E_y(\boldsymbol{y}|\boldsymbol{x}_n) \tag{7}$$
    together with some *stochastic noise*
$$\boldsymbol{n}_n \triangleq \boldsymbol{y}_n - f(\boldsymbol{x}_n) \tag{8}$$
  - stochastic noise is a characteristic of the environment and is independent of the NN
  - under this formulation, the goal of a NN is to learn $f(\boldsymbol{x})$ (7), from $\{(\boldsymbol{x}_n, \boldsymbol{y}_n)\}$
  - target function is assumed to be unknown
- Classification versus regression:
  - in a *classification* problem, $\{y_n\}$ is a finite set
  - in a *regression* problem, $\{y_n\}$ is a subset of the real number
    - regression means real-valued output

B. Datasets

- Obtaining large repository of data, along with talent, are the two scarce resources
  - this is a major bottleneck for startups
  - some products are launched with the purpose of collecting data rather than revenue
- Mean & variance:
  - consider sample data set $\{\boldsymbol{x}_n\}$, of *sample size* N
  - arrange the data set $\{\boldsymbol{x}_n\}$ into an $(N \times d)$ *data matrix* X, whose n$^{\text{th}}$ rows is $\boldsymbol{x}_n$

  - the mean $\bar{\boldsymbol{x}}$, associated with $\{\boldsymbol{x}_n\}$, is

$$\begin{aligned}\bar{\boldsymbol{x}} &= \frac{1}{N}\sum_{n\leq N}\boldsymbol{x}_n \\ &= \frac{1}{N}\mathbf{1}^T X\end{aligned} \tag{9}$$

    where $\mathbf{1}$ is $(N\times 1)$ vector of 1s
  - the covariance $\bar{C}$, associated with X is

$$\begin{aligned}\bar{C} &= \frac{1}{N}\sum_{n\leq N}(\boldsymbol{x}_n-\bar{\boldsymbol{x}})(\boldsymbol{x}_n-\bar{\boldsymbol{x}})^T \\ &= \frac{X^TX}{N}-\bar{\boldsymbol{x}}\bar{\boldsymbol{x}}^T\end{aligned} \tag{10}$$

- Dataset modifications:
  - various modifications can be applied to a dataset, including
    - preprocessing
    - normalization
    - expansion, or augmentation
    - reuse
- Data preprocessing:
  - dataset $\{\boldsymbol{x}_n\}$ is usually modified before applying to a NN, by making it zero mean,

$$\hat{\boldsymbol{x}}_n \triangleq \boldsymbol{x}_n - \bar{\boldsymbol{x}} \tag{11}$$

  - it is possible to reduce data dimensionality (d), using *principle component analysis*, or PCA (233)
  - a dataset $\{\boldsymbol{x}\}$ can be *whitened* by normalizing the variance on each component by dividing each component by the square root of its eigenvalue
  - for images, usually only mean centering is performed
- Batch normalization:
  - in practice the data-set is partitioned to *mini-batches* and the NN is updated one mini-batch at a time
    - denote the mini-batch size by m
  - with *batch normalization*, every layer k, of a network is augmented with another normalizing layer
  - each layer performs the following normalization:
    - determine the batch mean $\boldsymbol{\mu}_k$ (9) & variance $\boldsymbol{\sigma}_k^2$
    - given scale & shift parameters $(\gamma^{(k)}, \beta^{(k)})$ compute

$$\boldsymbol{x}_k \longrightarrow \hat{\boldsymbol{x}}_k \triangleq \tfrac{\boldsymbol{x}_k-\boldsymbol{\mu}_k}{\boldsymbol{\sigma}_k} \longrightarrow \boldsymbol{y}_k \triangleq \gamma^{(k)}\hat{\boldsymbol{x}}_k+\beta^{(k)} \tag{12}$$

- Data expansion or augmentation:
  - the training data $\{(\boldsymbol{x}_n, \boldsymbol{y}_n)\}$, can be artificially expanded by applying, on the available data, application-dependent transformations
    - for image recognition, can use
      - translation
      - rotation
      - horizontal flip
      - random crops & scales and then rescale
      - deformation, distortion, stretch and sheer
      - color jitter such as contrast, brightness, saturation
    - for speech recognition, can introduce background interference, change the speed etc.
- Data snooping:
  - if a data set has affected any step in the learning process, its ability to assess the outcome has been compromised
  - this phenomenon is known as *data snooping*
  - data reuse and expansion should be carried out carefully
  - data snooping is a common trap for a practitioner
- Data partitioning:
  - data is partitioned into training, testing and validation, see Section III

II. Fundamentals

A. Cost functions

- General remarks:
  - learning is accomplished by minimizing some cost function
    - or equivalently by reward maximization
  - cost functions are also referred to as *error metrics*, *lost functions*, or *objective functions*
- Pointwise cost function:
  - for a given input $\boldsymbol{x}$, the pointwise *cost function* between hypotheses h & f is

$$c_{h,y} \triangleq c\left(\boldsymbol{y}^{(h)}, \boldsymbol{y}^{(f)}\right) \tag{13}$$

    where
    - $\boldsymbol{y}^{(h)} \triangleq h(\boldsymbol{x})$ is the output associated with hypothesis h
    - such a cost function can be applied for all learning paradigms
    - when the label $\boldsymbol{y}$ associated with data $\boldsymbol{x}$ is available, the cost function associated with hypothesis h becomes

$$c_h \triangleq c\left(\boldsymbol{y}^{(h)}, \boldsymbol{y}\right) \tag{14}$$

- Cost function examples:
  1) The *Hamming distance*, defined as

$$c_{h,f} \triangleq [h(\boldsymbol{x}) \neq f(\boldsymbol{x})] \tag{15}$$

     is often used when $f(\boldsymbol{x}) \in \{0,1\}$
  2) The *generalized Euclidean distance* is defined as

$$c_{h,f} \triangleq (\boldsymbol{y}^{(h)} - \boldsymbol{y}^{(f)})^T Q\, (\boldsymbol{y}^{(h)} - \boldsymbol{y}^{(f)}) \tag{16}$$

     where Q is an arbitrary positive semi-definite matrix
     two special classes include
     - the *Euclidean distance*, when $Q = I$
     - the *Mahalanobis distance*, when Q is the inverse of the covariance matrix of the data
       - the Mahalanobis distance metric depends on the data set
       - the main advantage of the Mahalanobis distance over the standard Euclidean distance is that it takes into account correlations among the data dimensions and scale
  3) The log-likelihood cost function:
     - the log-likelihood cost function, from the ML criterion (238), is given by

$$\log_2 \frac{1}{P(y|\boldsymbol{x})} \tag{17}$$

     - since a NN can model $P(y|\boldsymbol{x})$ (5), using hypothesis h (4, 37), the log-likelihood cost function is expressed as

$$\begin{aligned} c_h &= \log_2 \frac{1}{h(\boldsymbol{x})} \\ &= -\log_2 h(\boldsymbol{x}) \end{aligned} \tag{18}$$

     - the log-likelihood cost function is often used with softmax activation (46)
  4) The *cross-entropy* cost function (246):
     - the cross-entropy measure is a useful metric when the NN is estimating the probability distribution associated with a classification problem
     - then the label $\boldsymbol{y}$ represents a probability distribution
       - $\boldsymbol{y} = p(\boldsymbol{x})$ is the desired probability distribution
       - $\hat{\boldsymbol{y}} = h(\boldsymbol{x})$ is the estimated probability distribution
     - from (246), the cross-entropy cost function associated with data $(\boldsymbol{x}, \boldsymbol{y})$ and hypothesis h is

$$c_h = -\sum_i y_i \log_2 h_i(\boldsymbol{x}) \tag{19}$$

     - with *disjoint classification* problems only a single outcome is valid, & $\boldsymbol{y}$ becomes a one-hot vector
       - the cross-entropy simplifies to

$$c_h \triangleq -\log_2 h_i(\boldsymbol{x}) \tag{20}$$

      * the above cost coincides with the log-likelihood cost function (18)

5) Multi-class SVM loss function:
   - consider a margin of $\triangle$, as described in Section XIII
   - given $(\boldsymbol{x}_i, y_i)$, define the score of the j^{th} class as

$$s_j \triangleq f_j(\boldsymbol{x}_i, \boldsymbol{w}), \tag{21}$$

   - the *multi-class SVM loss*, or *hinge loss*, for data $(\boldsymbol{x}_i, y_i)$ is defined as,

$$c_i \triangleq \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + \triangle) \tag{22}$$

6) The *cosine similarity metric*, $\in [1, 1]$, is defined as

$$c_{h,f} \triangleq \frac{\boldsymbol{y}^{(h)} \cdot \boldsymbol{y}^{(f)}}{\|\boldsymbol{y}^{(h)}\| \|\boldsymbol{y}^{(f)}\|} \tag{23}$$

7) The *Jaccard coefficient* is often used when the objects represent sets,

$$c(S1, S2) \triangleq \frac{|S1 \cap S2|}{|S1 \cup S2|} \tag{24}$$

- In-sample versus out-of-sample cost functions:
  - using (14), the *in-sample cost function* $C_{\text{in}}(h)$, over N samples, of using hypothesis h is defined as

$$C_{\text{in}}(h) \triangleq \frac{1}{N} \sum_{n=1}^{N} c(\boldsymbol{y}_n^{(h)}, \boldsymbol{y}_n) \tag{25}$$

  - the *out-of-sample* cost function $C_{\text{out}}(h)$ is defined as

$$C_{\text{out}}(h) \triangleq E_{x,y}[c(\, h(X),\, Y)] \tag{26}$$

B. Neuron models

- A neuron:
  - a neuron is also referred to as a *unit* or a *node*
  - a neuron takes a $(d+1)$ dimensional input vector $\boldsymbol{x}$, and outputs a scalar y
  - a neuron is some linear circuit followed by a non-linear function,

$$\boldsymbol{x} \xrightarrow{\ \boldsymbol{w}\ } s \triangleq \boldsymbol{w}^T \boldsymbol{x} \xrightarrow{\ \theta\ } y \triangleq \theta(s) \tag{27}$$

    * the linear portion is modeled with the parameter vector $\boldsymbol{w}$, called the *weight*
      · *linearity in the weights* means the weights have linear dependency
      · s is a scalar called the *signal* or the *score*
    * the non-linearity is introduced through the *activation function* θ
  - a neuron can also be defined in a more setting

$$y \triangleq \theta(\boldsymbol{x}, \boldsymbol{w}) \tag{28}$$

  - most neurons considered here have a $(d+1)$ dimensional $\boldsymbol{w}$, with $x^0 \triangleq 1 \Rightarrow w^0$ is the *bias*
  - a hypothesis h is a function of weights $\boldsymbol{w}$
- Non-linearities & Activation functions:
  - deeper networks can generate more sophisticated non-linear transformations
  - various activation functions can be applied on s (27)
  - in general, activations need to be monotonically non-decreasing, and mostly differentiable
- Feature transform:
  - in addition to activation functions, non-linearities could also be introduced by non-linear mapping of the input data
  - a *feature transform* Φ, is a non-linear transformation

$$\boldsymbol{x} = (x_0, \cdots, x_d) \xrightarrow{\ \phi\ } \boldsymbol{z} = (z_0, \cdots, z_{\tilde{d}}) \tag{29}$$

    where $\boldsymbol{z} \in \mathbb{R}^{\tilde{d}+1}$ is in the *feature space*
  - strictly speaking Φ may not satisfy the constraints of a function
  - the linear summation (27) generalizes to

$$\tilde{\boldsymbol{w}}^T \boldsymbol{z} = \tilde{\boldsymbol{w}}^T \Phi(\boldsymbol{x}) \tag{30}$$

  - linearity in weights is retained since non-linearities are applied on the inputs, not the weights,
  - if $\tilde{d} > d$, then we might have generalization concerns
  - if $\tilde{d} < d$, then we are acting as a learning machine, and should account for that for generalization
  - latest approaches rely on deeper NN rather than feature transformations to obtain sophisticated non-linearities
- Identity, linear classifiers & cybernetics:
  - in certain networks, a fraction of the neurons do not have activation functions
  - in this case, θ collapses to the identity function, and the *linear regression* hypotheses is obtained, see Section XV

$$\theta(s) = s = \boldsymbol{w}^T\boldsymbol{x} \tag{31}$$

  - if a single neuron is used, a hypothesis can then be determined by the $(d+1)$ parameters $\boldsymbol{w} = \{w_i\}$, and the signal s (27) itself is used to classify inputs
    * *cybernetics* implied learning $\boldsymbol{w}$ of such a model
- List of activation functions:
  1) Perceptron:
     - the *McCulloch-Pitts Neuron*, 1943, is a binary classifier, obtained by setting the activation function to be the sign() of the input value

$$\theta_{MP}(s) \triangleq \text{sign}(s) = \text{sign}(\boldsymbol{w}^T\boldsymbol{x}) \tag{32}$$

     - a *perceptron* [2] is a McCulloch-Pitts Neuron that can learn its weights
     - since the derivative of this activation vanishes, it is not a useful activation for learning
     - based on Section XI a complexity measure can be associated with the perceptron
       * theorem: $d_{VC} = d + 1$, which coincides with the number of parameters w_i
  2) The *softsign* function is the soft version of the McCulloch-Pitts Neuron:

$$\theta_{SS}(s) \triangleq \frac{s}{1+|s|}, \tag{33}$$

  3) The logistic sigmoid function:
     - the *logistic sigmoid* function:

$$\theta_L(s) \triangleq \frac{e^s}{1+e^s} = \frac{1}{1+e^{-\boldsymbol{w}^T\boldsymbol{x}}} \tag{34}$$

     - $\theta_L(s) \geq 0 \Rightarrow x_i \geq 0$, (i.e. not centered around zero) implies the changes of w_i, due to gradient, are all positive or negative, see sub-section II-C
     - furthermore, saturated regions have no gradient information
     - useful for gating functions
  4) Stochastic neuron & logistic regression:
     - a boolean variable $y \in \{0, 1\}$ can be referred to as an *indicator variable*
     - in *logistic regression*, the goal is to determine the conditional probability of an indicator variable, $P(y = 1|\boldsymbol{x})$
     - since the output of (34) is $\in [0, 1]$, this output can be reinterpreted as a conditional probability, $P(y = 1|\boldsymbol{x})$
     - a *stochastic neuron* has the same activation function as the the logistic sigmoid, $\theta_L(s)$ that implements a logistic regression
       * the goal is to estimate $P(y = 1|\boldsymbol{x})$ rather than y
     - this is similar to obtaining soft data in data communication
     - in general $P(y = 1|\boldsymbol{x})$ is not a linear function with respect to $\boldsymbol{x}$
     - logistic regression converts the problem to a linear regression (with infinite range), by *assuming* the transformation

$$\begin{aligned}\log\frac{P(y=1|\boldsymbol{x})}{1-P(y=1|\boldsymbol{x})} &= \boldsymbol{w}^T\boldsymbol{x} = s \Rightarrow \\ P(y=1|\boldsymbol{x}) &= \frac{e^s}{1+e^s},\end{aligned} \tag{35}$$

     - note that this coincides with (34), and that

$$\begin{aligned}\theta(-s) &= \frac{1}{1+e^s} \\ &= 1 - \frac{e^s}{1+e^s} \\ &= 1 - \theta(s) \\ &= P(y=0|\boldsymbol{x})\end{aligned} \tag{36}$$

- both $P(y=1|\boldsymbol{x})$ & $P(y=0|\boldsymbol{x})$ can be expressed by the unified form

$$P(y|\boldsymbol{x}) = \theta(y\boldsymbol{w}^T\boldsymbol{x}) \tag{37}$$

5) The *tangent hyperbolic* function:

$$\theta_T(s) \triangleq \tanh(s) = \frac{e^s - e^{-s}}{e^s + e^{-s}} \tag{38}$$

- popular in 1980s
- the range is $\theta_T \in [-1, 1]$
- $\theta_T(s)$ can be obtained from θ_L (34) by the transformation $y = 2x - 1$

6) The *hard tanh* function:

$$\theta_{HT}(s) \triangleq \begin{cases} -1, & \text{if } s < -1 \\ s, & \text{if } -1 \le s \le 1 \\ 1, & \text{if } s > 1 \end{cases} \tag{39}$$

7) The *rectified linear unit*, or ReLU:

$$\begin{aligned} \theta_R(s) &\triangleq \max(0, s) \\ &= \max(0, \boldsymbol{w}^T\boldsymbol{x}) \end{aligned} \tag{40}$$

- currently a popular activation function
- typically learns faster in networks with many layers
- when the signal stays negative, we have a *dead ReLU*
  - ∗ positive bias initialization addresses the dead ReLU concern

8) The *parametric ReLU*, or PReLU:

$$\theta_{PR}(s) \triangleq \max(\alpha s, s) \tag{41}$$

- setting $\alpha = 0.01$ gives the *leaky ReLU*:

$$\theta_{LR}(s) \triangleq \max(0.01s, s) \tag{42}$$

9) The *exponential linear unit*, or *ELU*:

$$\theta_{ELU}(s) \triangleq \begin{cases} s, & \text{if } s > 0 \\ \alpha(e^s - 1), & \text{if } s \le 0 \end{cases} \tag{43}$$

10) The *softplus* function is smooth (soft) version of (40):

$$\theta_{SP}(s) \triangleq \log(1 + \exp(x)) \tag{44}$$

11) The *Maxout* activation:

$$\theta_{MO}(\boldsymbol{x}) \triangleq \max(\boldsymbol{w}_1^T\boldsymbol{x}, \boldsymbol{w}_2^T\boldsymbol{x}) \tag{45}$$

12) The *softmax* function, with $1 \le j \le K$ output nodes, is

$$\theta_{SM}(j; s_1^K) \triangleq \frac{e^{s_j}}{\sum_{k=1}^{k=K} e^{s_k}} \tag{46}$$

where the j^{th} node generates the probability of the j^{th} event

- since $\theta(j) \ge 0$ and $\sum_j \theta(j) = 1$, we can also interpret $\{\theta(j)\}$ as probabilities
- softmax can be useful with classification problems involving disjoint classes
- the log-likelihood cost function is often used with softmax activation

- Log-likelihood criterion on stochastic neuron:
  - substituting (37) into (18)

$$\begin{aligned} c_{\boldsymbol{w}} &= \log \frac{1}{\theta(y_{\boldsymbol{w}}^T\boldsymbol{x})} \\ &= \log(1 + e^{-y_{\boldsymbol{w}}^T\boldsymbol{x}}) \end{aligned} \tag{47}$$

  - analytic solution to $\nabla C_{\text{in}}(\boldsymbol{w}) = 0$ is not feasible, but gradient descent algorithms can be used
  - with classification problems, this cost metric converges faster than the mean square error metric

C. Update Rules

- Perceptron Learning Algorithm (PLA):
  - pick any $(\boldsymbol{x}, y)$ that is misclassified, and make the update

$$\begin{aligned} &\boldsymbol{w}(t+1) = \boldsymbol{w}(t) + y\boldsymbol{x} \Rightarrow \\ &y\,\boldsymbol{w}^T(t+1)\boldsymbol{x} = y\,\boldsymbol{w}^T(t)\boldsymbol{x} + y^2\|x\|^2 \Rightarrow \\ &y\,\boldsymbol{w}^T(t+1)\boldsymbol{x} > y\,\boldsymbol{w}^T(t)\boldsymbol{x} \end{aligned} \tag{48}$$

  - since its a misclassified pair

$$y\,\boldsymbol{w}^T(t)\boldsymbol{x} < 0 \tag{49}$$

  - combining (48) with (49) implies $\boldsymbol{w}$ is moving in the right direction
  - theorem: if the input data is linearly separable then the PLA will converge with $C_{\text{in}} = 0$
- Pocket algorithm:
  - when input data-set is not linearly separable, then the PLA is not stable
    * the problem of choosing $\boldsymbol{w}$ that minimizes C_{in}, i.e. error rate, is NP-hard
  - the *pocket algorithm* is a variation of PLA, that keeps "in its pocket" the best $\boldsymbol{w}$
    * in other words, pick the best $\boldsymbol{w}$ rather than the last $\boldsymbol{w}$
    * C_{in} needs to be computed in every step
- Gradient descent:
  - gradient descent can be applied to a differentiable graph whether or not there is an analytic solution
  - let $\nabla C_{\text{in}}(\boldsymbol{w}) = \nabla_\omega C_{\text{in}}(\boldsymbol{w})$ be the gradient of the cost function $C_{\text{in}}(\boldsymbol{w})$ with respect to $\boldsymbol{w}$
  - gradient descent is based on the observation that $C_{\text{in}}(\boldsymbol{w})$ decreases fastest in the direction of $-\nabla C_{\text{in}}(\boldsymbol{w})$
  - define $\hat{\boldsymbol{v}}$ to be a unit vector in the direction of *steepest descent*

$$\hat{\boldsymbol{v}} \triangleq -\frac{\nabla C_{\text{in}}(\boldsymbol{w})}{\|\nabla C_{\text{in}}(\boldsymbol{w})\|} \tag{50}$$

  - *gradient descent* is an iterative solution, where at time $(t+1)$, the weights are adjusted as

$$\begin{aligned} \boldsymbol{w}(t+1) &= \boldsymbol{w}(t) + \delta\boldsymbol{w} \\ &= \boldsymbol{w}(t) + \eta_t\hat{\boldsymbol{v}} \end{aligned} \tag{51}$$

    where η_t is a hyper-parameter called the *learning rate*
  - to see why gradient descent proceeds in direction of $-\nabla C_{\text{in}}(\boldsymbol{w})$, apply Taylor expansion on $C_{\text{in}}(\boldsymbol{w})$

$$\begin{aligned} \Delta C_{\text{in}} &\triangleq C_{\text{in}}(\boldsymbol{w}(t) + \delta\boldsymbol{w}) - C_{\text{in}}(\boldsymbol{w}(t)) \\ &\approx \nabla C_{\text{in}}(\boldsymbol{w}(t)) \cdot \delta\boldsymbol{w} \end{aligned} \tag{52}$$

  - choosing $\delta\boldsymbol{w} = \hat{\boldsymbol{v}}$ and $\eta_t = 1$

$$\begin{aligned} \Delta C_{\text{in}} &= \nabla C_{\text{in}}(\boldsymbol{w}(t)) \cdot \hat{\boldsymbol{v}} \\ &= -\nabla C_{\text{in}}(\boldsymbol{w}(t)) \cdot \frac{\nabla C_{\text{in}}(\boldsymbol{w}(t))}{\|\nabla C_{\text{in}}(\boldsymbol{w}(t))\|} \\ &\geq -\|\nabla C_{\text{in}}(\boldsymbol{w}(t))\| \end{aligned} \tag{53}$$

    * steepest descent is achieved when we have equality in (52), i.e. when

$$\delta\boldsymbol{w} = \hat{\boldsymbol{v}} = -\frac{\nabla C_{\text{in}}(\boldsymbol{w}(t))}{\|\nabla C_{\text{in}}(\boldsymbol{w}(t))\|} \tag{54}$$

- Local and global minima:
  - if the cost function is *convex* then there is no *local minima* but only a single *global minimum*
  - in practice, poor local minima are rarely a problem with large networks, or in higher dimensional space
  - instead, the landscape is packed with a combinatorially large number of saddle points where the gradient is zero in many but not all dimensions
  - practically, it does not much matter which of these saddle points the algorithm ends up being
- Batch gradient descent:
  - it is desirable to get large steps when we are far from minimum, & small steps when close to the minimum
  - can choose learning rate η_t to be proportional to the gradient

$$\eta_t = \eta\|\nabla C_{\text{in}}(\boldsymbol{w}(t))\| \tag{55}$$

- substituting (54, 55) into $\delta \boldsymbol{w}(t)$

$$\begin{aligned}\delta \boldsymbol{w}(t) &= \eta_t \, \hat{\boldsymbol{v}} \\ &= -\eta \|\nabla C_{\text{in}}(\boldsymbol{w}(t))\| \frac{\nabla C_{\text{in}}(\boldsymbol{w}(t))}{\|\nabla C_{\text{in}}(\boldsymbol{w}(t))\|} \\ &= -\eta \nabla C_{\text{in}}(\boldsymbol{w}(t))\end{aligned} \tag{56}$$

- it follows that

$$\boxed{\boldsymbol{w}(t+1) = \boldsymbol{w}(t) - \eta \, \nabla C_{\text{in}}(\boldsymbol{w}(t))} \tag{57}$$

- this algorithm is sometimes referred to as the *batch gradient descent* or *fixed learning rate gradient descent algorithm*
- note that a single update is generated from the entire data set
  - batch gradient descent can be very slow since the gradients for the whole dataset are calculated to perform just one update
  - batch gradient descent are intractable for datasets that don't fit in memory

- Batch gradient descent for logistic regression:
  - the gradient of (47) is

$$\begin{aligned}\nabla C_{\text{in}}(\boldsymbol{w}(t)) &= -\frac{1}{N} \sum_{n=1}^{N} \frac{y_n \boldsymbol{x}_n e^{-y_n \boldsymbol{w}^T \boldsymbol{x}_n}}{(1 + e^{-y_n \boldsymbol{w}^T \boldsymbol{x}_n})} \\ &= -\frac{1}{N} \sum_{n=1}^{N} \frac{y_n \boldsymbol{x}_n}{1 + e^{y_n \boldsymbol{w}^T \boldsymbol{x}_n}} \\ &= -\frac{1}{N} \sum_{n=1}^{N} y_n \boldsymbol{x}_n \theta(-y \boldsymbol{w}^T \boldsymbol{x}_n)\end{aligned} \tag{58}$$

  - substituting (58) into (56)

$$\delta \boldsymbol{w}(t) = \frac{1}{N} \sum_{n=1}^{N} y_n \boldsymbol{x}_n \theta(-y \boldsymbol{w}^T \boldsymbol{x}_n) \tag{59}$$

- Stochastic gradient descent (SGD):
  - *stochastic gradient descent* (SGD), is the sequential version of batch gradient descent, where an update is generated for each sample point $(\boldsymbol{x}_n, y_n)$
    - $(\boldsymbol{x}_n, y_n)$ is picked uniformly at random, hence the name 'stochastic'
  - compared to batch gradient descent, SGD is simpler, faster and, due to randomization, is less prone to getting trapped in a local minima
  - the rule of thumb is to use $\delta \approx 0.1$
  - SGD complicates convergence to the exact minimum, as SGD will keep overshooting
- Mini-batch gradient descent:
  - *mini-batch gradient descent* implies a single gradient update is generated per mini-batch of data $\{(\boldsymbol{x}_n, y_n)\}$
  - each element of the mini-batch is picked uniformly at random
  - mini-batch gradient descent takes the best of both SGD and batch gradient descent
    - compared to SGD, it reduces the variance of the parameter updates, which can lead to more stable convergence
  - common mini-batch sizes range between 50 and 256
  - the term SGD usually is employed also when mini-batches are used
- Learning rate with annealing:
  - with *step decay*, the learning rate is reduced by some factor as a function of training iterations
  - the *exponential decay* and the $1/t$-*decay* annealing functions are respectively

$$\begin{aligned}\eta_t &= \eta_o \exp(-kt) \\ \eta_t &= \frac{\eta_o}{1 + kt}\end{aligned} \tag{60}$$

  where η_0, k are hyper-parameters & t is the iteration number
- Hessian technique:
  - applying Taylor's extension, as in (52), but including second order terms

$$\Delta C_{\text{in}}(\boldsymbol{w} + \delta \boldsymbol{w}) \approx \; \nabla C_{\text{in}} \cdot \delta \boldsymbol{w} \; + \; \frac{1}{2} \delta \boldsymbol{w}^T H \delta \boldsymbol{w} \tag{61}$$

where the *Hessian matrix* H, is defined as

$$H_{jk} \triangleq \frac{\partial^2 C_{\text{in}}}{\partial w_j \partial w_k} \tag{62}$$

- differentiating (61) with respect to $\delta \boldsymbol{w}$, and setting it to zero

$$\begin{aligned} 0 &= \nabla C_{\text{in}} + H\delta\, \boldsymbol{w} \Rightarrow \\ \delta \boldsymbol{w} &= -H^{-1}\nabla C \end{aligned} \tag{63}$$

- this approach to minimizing a cost function is known as the *Hessian technique* or the *Hessian optimization*
  - * it is also referred to as the *Newton's method*
- Hessian methods converge in fewer steps than standard gradient descent
- furthermore, the Hessian approach could avoid many pathologies that can occur in gradient descent
- these benefits come at the expense of significant additional complexity
- two simplifications are BGFS and limited-memory BFGS (L-BFGS)

• Momentum based gradient descent:
- the momentum technique modifies gradient descent by introducing two new concepts:
  1) Introducing the notion of ”velocity” $\boldsymbol{v}_t$, for the parameters we're trying to optimize
     * the gradient acts to change the velocity, not (directly) the ”position” $\boldsymbol{w}_t$, in much the same way as physical forces change the velocity, that only indirectly affect position
  2) Introduces the notion of ”friction” which tends to gradually reduce the velocity.
- the update rules are summarized as follows

$$\begin{aligned} \boldsymbol{v}_t &= \mu \boldsymbol{v}_{t-1} - \eta\, \nabla C \\ \boldsymbol{w}_t &= \boldsymbol{w}_{t-1} + \boldsymbol{v}_t \end{aligned} \tag{64}$$

where μ is the *momentum coefficient*, that controls the amount of damping or friction in the system
  * $\mu = 0$ corresponds to the gradient descent method
  * $0.5 \le \mu \le 0.9$ is common
- the momentum technique is more practical than the Hessian technique
- the momentum technique often speeds up learning and is commonly used

• Nesterov accelerated gradient:
- *Nesterov accelerated gradient* (NAG), is a momentum based technique, where the gradient is evaluated at $\boldsymbol{w} + \mu \boldsymbol{v}$ rather than at $\boldsymbol{w}$

• Adagrad:
- *Adagrad* adapts the learning rate to the parameters
- so far, at time t, all the parameter w_i used the same learning rate
- define G_t to be a diagonal matrix, where $G_{t,ii}$ is the sum of the squares of the gradients w.r.t. w_i up to time step t
- the update rule for Adagrad is

$$w_{t,i+1} = w_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \nabla C(w_i) \tag{65}$$

where $\epsilon \sim 10^{-8}$ is a smoothing term that avoids division by zero
- compared to SGD, the learning rate is adaptively scaled by $\sqrt{G_{t,ii} + \epsilon}$
- without the square root operation, the algorithm performs much worse
- Adagrad eliminates the need to manually tune the learning rate
- most implementations use a default value of $\eta = 0.01$
- Adagrad also implicitly implements momentum based updates (why?)
- Adagrad's main weakness is its accumulation of the squared gradients in the denominator which eventually makes infinitesimally small

• Adadelta:
- *Adadelta* is an extension of Adagrad that seeks to reduce its aggressive, monotonically decreasing learning rate
- instead of accumulating all past squared gradients, Adadelta restricts the window of accumulated past gradients to some fixed size L

• RMSProp:
- *RMSProp* is an adaptive gradient technique where the learning rate is tailored for each parameter

  - if the gradient at time t is $\nabla_t C$, then the learning rate, at time t is

$$\begin{aligned} s_t &= \alpha s_{t-1} + (1-\alpha)\|\nabla_t C\|^2 \\ \boldsymbol{w}_t &= \boldsymbol{w}_{t-1} + \eta \frac{\nabla_t C}{\sqrt{s_t}} \end{aligned} \tag{66}$$

- Adam:
  - in the momentum technique (64), $\boldsymbol{v}_t$ is linear in $\boldsymbol{g}_t$, whereas in RMSProp (66, s_t is quadratic in $\boldsymbol{g}_t$
  - *Adam* combines features from both of these techniques

$$\begin{aligned} \boldsymbol{v}_t &= \mu \boldsymbol{v}_{t-1} + (1-\mu)\nabla_t \\ s_t &= \alpha s_{t-1} + (1-\alpha)\|\nabla_t\|^2 \\ \boldsymbol{w}_t &= \boldsymbol{w}_{t-1} + \eta \frac{\boldsymbol{v}_t}{\sqrt{s_t}} \end{aligned} \tag{67}$$

  - its is recommended to use $\mu = 0.9$, $\alpha = 0.999$

III. Generalization

- Generalization & feasibility of learning:
  - *generalization* is the capability of predicting a function for unobserved data
    * learning an arbitrary boolean function is not feasible by just observing a subset of the function
  - in machine learning, generalization is done based on probabilistic inference, Section I-A
  - learning is feasible under the assumption that the training & testing samples share the same distribution
- Two goals of learning: choose hypothesis $g \in \{h_i\}$:
  1) to *approximate*, or minimize $C(h)_{\text{in}}$ (25), and
  2) to generalize, i.e. from (25, 26),

$$C(g)_{\text{out}} \approx C(g)_{\text{in}} \tag{68}$$

     where $\approx$ means $C_{\text{in}}(h) = C_{\text{out}}(h)$ is *probably approximately correct*, or P.A.C.
- Testing:
  - *testing* is performed after a hypothesis g is chosen
  - testing is performed as a *final* evaluation of accuracy
  - testing is performed on a different data set than the training set
- Overfitting:
  - *overfitting* is the phenomenon where the in-sample data is fitted more than is warranted
  - small C_{in} is observed at the expense of C_{out}
  - when overfitting, the network is learning about the peculiarities of the training set
  - when overfitting, the network is
    * more memorizing than learning
    * not learning to generalize from trend
  - a simple approach to contain overfitting is to *early termination*, where training stops if no improvements are observed
- Overfitting dependencies:
  - overfitting gets worst as
    * stochastic noise variance σ_n^2 increases
    * target function complexity increases
      · for example if target function is modeled by a polynomial of higher degree
    * data size N decreases
      · In many situations the data size should decide the complexity of the hypothesis family chosen, not the assumed complexity of the target function
- Deterministic noise:
  - as defined in (190), deterministic noise is the part of the target function f that $\mathcal{H}$ cannot capture
  - deterministic noise is defined as

$$f(\boldsymbol{x}) - g(\boldsymbol{x}) \tag{69}$$

  - deterministic noise occurs when
    * $\mathcal{H}$ does not capture important aspects of f, i.e. $\mathcal{H}$ is oversimplified, or when
    * target function complexity Q_f is large and N is not large enough

- unlike stochastic noise, deterministic noise
  - depends on $\mathcal{H}$, and
  - is repeatable
- given $\mathcal{H}$, both noise sources have similar impact on the learning process

- VC generalization bound:
  - VC bound provide a lower bound on $C_{\text{out}}(g) - C_{\text{in}}(g)$ as a function of sample size N
  - see Appendix XI, and (188)
  - in general, the VC bounds are very loose
- Bias & variance model:
  - an alternative method to decompose $C_{\text{out}}(g)$ is the *bias & variance method* reviewed in Appendix XII
- Two common approaches to address overfitting:
  1) Regularization
  2) Validation

A. Regularization

- General remarks:
  - *regularization* constrains the learning algorithm to improve out-of-sample error
    - constraining implies lowering d_{VC}
  - regularization is as much an art as it is science
    - there is no systematic understanding, only heuristics
  - in general, regularization reduces noise more than it limits the signal
- The augmented metric:
  - the goal of regularization is to minimize an *augmented metric*

$$C_{\text{aug}}(h) = C_{\text{in}}(h) + \Omega(h) \tag{70}$$

    rather than $C_{\text{in}}(h)$ alone, see (187)
  - $\Omega(h)$ is the *regularizer*, or the *regularization term*
    - $\Omega(h)$ depends on h rather than $\mathcal{H}$
    - $\Omega(h)$ is a proxy for overfit penalty
  - the augmented metric $C_{\text{aug}}(h)$ is a proxy for $C_{\text{out}}(h)$
- Regularization methods:
  1) L2 regularization, or weight decay
     - other quadratic forms include
       - L2 regularization with importance factors
       - Tikhonov regularization
  2) L1 regularization
  3) Soft weight elimination
  4) Soft targets to train
  5) Dropout
  6) Artificially expand the data set
- Hard & soft order constraint:
  - for definiteness, consider a linear regression problem on the $\mathcal{Z}$ space (30), of polynomials
  - a polynomial hypothesis with lower degree (simpler) may result in better out-of-sample error
    - excluding all polynomials higher than a degree threshold is known as *hard order constraint*
  - an example of a *soft order constraint* is the constraint that polynomial coefficients satisfy

$$\sum_{q=0}^{Q} w_q^2 \leq C \Rightarrow \boldsymbol{w}^T\boldsymbol{w} \leq C \tag{71}$$

    - this is similar to FIR with normalized weight energy constraint
- L2 regularization:
  - the augmented error metric associated with (71) uses Lagrange multipliers to solve this constrained linear regression problem

$$C_{\text{aug}}(\boldsymbol{w}) = C_{\text{in}}(\boldsymbol{w}) + \frac{\lambda}{N}\boldsymbol{w}^T\boldsymbol{w} \tag{72}$$

- with this augmented metric, the network prefers to learn small weights, all other things being equal
  - $*$ the smallness of the weights means that the behaviour of the network won't change too much if we change a few random inputs here and there
  - $*$ this implies less noisy behaviour and simpler models
- λ is called the *regularization parameter*
- minimizing (72) is called *L2 regularization*, or *weight decay*, since taking the gradient of (72) gives the additional term that is proportional to $\boldsymbol{w}$
- for the linear regression case, the resulting weight vector becomes (compare to (249))

$$\boldsymbol{w}_{\text{reg}} = (Z^T Z + \lambda I)^{-1} Z^T \boldsymbol{y} \tag{73}$$

- for gradient descent, the updated weights have the form

$$w \to \left(1 - \frac{\eta\,\lambda}{N}\right) w - \eta \frac{\partial e}{\partial w} \tag{74}$$

- Weight decay of bias terms:
  - having a large bias doesn't make a neuron sensitive to its inputs
  - no need to worry about large biases enabling our network to learn the noise in our training data
  - allowing large biases gives our networks more flexibility in behaviour
    - $*$ in particular, large biases make it easier for neurons to saturate, which is sometimes desirable
  - for these reasons, usually bias terms are not regularized
- Alternative weight constraints:
  - the constraint (71) can be generalized to,

$$\sum_{q=0}^{Q} \gamma_q w_q^2 \le C \tag{75}$$

    where the *importance factor* $\{\gamma_q\}$ are parameters that amplify/attenuate certain degrees
  - in neural networks, different layers may get different importance factors
  - the *Tikhonov regularizer*

$$\boldsymbol{w}^T \Gamma^T \Gamma \boldsymbol{w} \le C \tag{76}$$

    considers non-diagonal quadratic form, and captures relationships within w_i by using matrix Γ
- L1 regularization:
  - the augmented error metric of L2 regularization (72), is modified to

$$C_{\text{aug}}(\boldsymbol{w}) = C_{\text{in}}(\boldsymbol{w}) + \frac{\lambda}{N} \sum_n |w_n| \tag{77}$$

  - the weights are then updated as

$$w \to w - \frac{\eta\,\lambda}{N} \text{sign}(w) - \eta \frac{\partial e}{\partial w} \tag{78}$$

  - since derivative is not defined at $w = 0$, apply the un-regularized rule for stochastic gradient, i.e. set $\text{sign}(0) = 0$
  - compared to (74), the decay in weight is constant, whereas in L2 regularization it is proportional w
  - L1 regularization tends to concentrate the weight of the network in a relatively small number of high-importance connections, while the other weights are driven toward zero
- Hard & soft weight elimination:
  - with *hard weight elimination*, weights with values less than a threshold are eliminated
  - with *soft weight elimination*, the regularizer could be defined as

$$\Omega(\boldsymbol{w}) \triangleq \sum_{i,j,l} \frac{(w_{ij}^{(l)})^2}{\beta^2 + (w_{ij}^{(l)})^2} \tag{79}$$

    - $*$ for $\|\boldsymbol{w}^{(l)}\| \ll \beta$, numerator dominates and the regularizer reduces to weight decay (72)
    - $*$ for $\|\boldsymbol{w}^{(l)}\| \gg \beta$, $\Omega(\boldsymbol{w})$ is a constant and no updates are generated from the regularizer
    - $*$ in summary, the regularizer attenuates small weights but not the large weights
- Dropout & dropconnect:
  - *dropout* modifies the network itself rather than modifying the cost metric

    - more specifically, for each training example, forward propagation involves randomly deleting any hidden neuron with probability p
        * the error is then backpropagated only through the remaining activations
        * $p = 0.5$ is common
    - we can think of dropout as a way of making sure that the model is robust to the loss of any individual piece of evidence
    - dropout has been useful in training large, deep networks, where the problem of overfitting is often acute
    - inverted dropout: after dropout the signal levels should be attenuated by p, (not $\sqrt{p}$?) since more inputs are contributing
    - *dropconnect* generalizes dropout by randomly dropping the weights rather than the activations
- Soft targets:
    - *soft target* means that, the label is a probability distribution rather than a one-hot vector
    - when using a small network, use some combination of soft target, obtained from a larger circuit, and hard target to train data,
    - soft targets are good regularizers
- Practical rule for regularization:
    - constrain learning towards smoother hypothesis, since noise is high frequency
    - in general smaller weights result in smoother hypothesis

B. Validation

- The concept:
    - partition available training data, that was originally designated to training, to two parts

$$\mathcal{D} = \mathcal{D}_{\text{val}} \cup \mathcal{D}_{\text{train}} \tag{80}$$

      where the validation sample size is $|\mathcal{D}_{\text{val}}| = K$, and the re-allocated *training* data size as $|\mathcal{D}_{\text{train}}| = N - K$
        * validation comes at the expense of training
        * validation is used to optimize various hyper-parameters
    - similar to the in-sample error (25), validation error is defined as

$$C_{\text{val}}(h) \triangleq \frac{1}{K} \sum_{n=1}^{K} e(h(\boldsymbol{x}_n), y_n) \tag{81}$$

    - whereas regularization predicts C_{out} indirectly through $\Omega(h)$, *validation* directly measures C_{out}
- A validation procedure:
    - generate a hypothesis g^- from $N - K$ samples
    - measure $C_{\text{out}}(g^-)$ from remaining K samples
    - generate a hypothesis g from all N samples
    - report g with measure $C_{\text{out}}(g^-)$
    - as a rule of thumb, use $K \approx \frac{N}{5}$
- Early stopping:
    - with *early stopping*, validation data is used to monitor overfitting, picking the hypothesis where C_{out} estimate is minimized
    - validation then becomes an integral part of training
- Training, testing and validating:
    - when training, C_{in} is optimistically (deceptively) biased
    - since testing does not alter the outcome of the learning process, C_{test} is unbiased
    - validating is similar to testing but can influence the outcome of the learning process because of feedback
        * then validation becomes optimistically biases as in training
        * i.e. C_{val} becomes a biased estimate of C_{out}
    - we want C_{val} to be only slightly contaminated to be useful
- Dependence of $g \in \mathcal{H}$ on data:
    - g is a strong function of peculiarities of the training data
    - g should be a weak function of the peculiarities of validation data
    - g is independent of the peculiarities of the test data
- M models:

  - many validation approaches can be interpreted as choosing one of M hypotheses, where each hypothesis was obtained from some model and training data
  - then validation coincides with training, and bounds similar to (179, 184) can be applied
- Cross-validation:
  - so far the goal was to achieve

$$C_{\text{out}}(g) \approx C_{\text{out}}(g^{-}) \approx C_{\text{val}}(g^{-}) \tag{82}$$

    where for the first approximation to hold, K needed to be small, while for the second approximation K needed to be large
  - in theory can run training N separate times, where at each attempt the n^{th} sample point is removed for validation purposes
  - the *cross-validation error* is then defined as the average of the N validation errors

$$E_{\text{cv}} \triangleq \frac{1}{N}\sum_{n=1}^{N} e_n \tag{83}$$

  - even though
    * each error term is from a different hypothesis, and
    * the error terms are correlated

    effectively, the error samples are almost uncorrelated
  - in summary, effectively $K = 1$ was used for first approximation of (82), and $K = N$ was used for second
- R-fold validation:
  - cross validation, is not practical as described above, since it introduces N times amplification in training time
  - *R-fold validation* implies partitioning sample data into R subsets rather than N
  - training sessions reduce from N to R
  - rule of thumb is to use 10-fold cross-validation

IV. Feedforward Networks with Supervision

- Historical overview:
  - *cybernetics* in the 1940s1960s
    * biological learning
    * perceptron
  - artificial neural networks (ANN) or *connectionism*, in the 1980s-1990s
    * parallel distributed processing (Rumelhart et al., 1986 ; McClelland et al., 1995)
    * a large number of simple computational units can achieve intelligent behavior when networked together
    * back propagation (Rumelhart et al., 1986)
  - *deep learning* starting in 2006, see Section IV-D
    * pioneered by Georey Hinton at University of Toronto, Yoshua Bengio at University of Montreal, and Yann LeCun at New York University
- Universality:
  - since a perceptron can model a NAND, a network of perceptrons can model any boolean function
  - consider an arbitrary continuous function $f(\boldsymbol{x})$, with range within the activation function
  - then for any $\epsilon > 0$, there exist a NN $g(x)$, with a single hidden layer, such that

$$|g(\boldsymbol{x}) - f(\boldsymbol{x})| < \epsilon \tag{84}$$

    for all inputs $\boldsymbol{x}$
  - the underlying concept is for the hidden layers to generate a basis function
  - the result holds for any activation function $\theta(z)$, where $\theta(\pm\infty)$ are distinct and well-defined
  - the question is not whether any particular function is computable, but rather what's a good way to compute the function
- Feedforward networks:
  - feedforward multilayer networks are the most popular ANNs
  - such networks include an *input layer*, an *output layer*, and *hidden layers*
  - the role of the *hidden layers* can be interpreted as distorting the input in a non-linear way so that categories become linearly separable by the last layer

A. *Multilayer perceptrons*

- General remarks:
  - *multilayer perceptron*, or MLP, is a feedforward network where each layer is fully connected (FC) to the previous layer
  - proposed in the 1980s
  - the name is misnomer since the unit of MLP is not the perceptron
  - MLPs use the supervised backpropagation algorithm to learn as will be discussed shortly
  - MLPs do not learn well in the presence of many hidden layers
- Notation:
  - a network is L layers with $l \in \{0, 1, \cdots, L\}$
    * the input layer corresponds to $l = 0$ and does not count as a layer
    * the output layer corresponds to $l = L$ which determines the value of the function
  - layer l has *dimension* $d^{(l)}$, which means it has $(d^{(l)} + 1)$ *nodes* (or *units*)
    * every layer has a *bias node*, labelled by 0
  - connections are between layers $(l-1)$ and l
    * $\boldsymbol{W}^{(l)} = \{w_{ij}^{(l)}\}$ is the weight from node i in layer $(l-1)$ to node j in layer l
    * the signal and the output of node j in layer l are denoted respectively as $\boldsymbol{s}^{(l)} = \{s_j^{(l)}\}$ and $\boldsymbol{x}^{(l)} = \{x_j^{(l)}\}$
  - hypothesis $h(\boldsymbol{w}) \in \mathcal{H}$ corresponds to hypothesis

$$\boldsymbol{w} \triangleq \{\boldsymbol{W}^{(1)}, \boldsymbol{W}^{(2)}, \cdots \boldsymbol{W}^{(L)}\} \tag{85}$$

- Forward propagation:
  - load $\boldsymbol{x}^{(0)} = \boldsymbol{x}$
  - for $l \in \{1, 2, \cdots, L\}$

$$\begin{aligned} \boldsymbol{s}^{(l)} &= (\boldsymbol{W}^{(l)})^T \boldsymbol{x}^{(l-1)} \\ \boldsymbol{x}^{(l)} &= \begin{bmatrix} 1 \\ \theta(\boldsymbol{s}^{(l)}) \end{bmatrix} \end{aligned} \tag{86}$$

  - the final output is $h(\boldsymbol{x}) = \boldsymbol{x}^{(L)}$
- The sensitivity vector:
  - the *sensitivity vector* $\boldsymbol{\delta}^{(l)}$ has components defined as

$$\boxed{\boldsymbol{\delta}_j^{(l)} \triangleq \frac{\partial C_{\text{in}}}{\partial \boldsymbol{s}_j^{(l)}}} \tag{87}$$

    where the signal $\boldsymbol{s}_j^{(l)}$ is given by (86), and the sample cost function C_{in} is defined in (25)
  - interpretation
    * $\boldsymbol{\delta}_j^{(l)}$ is the 'signal error' in the j^{th} unit
    * $\|\boldsymbol{\delta}^{(l)}\|$ is a measure on how fast layer l is learning
  - applying the chain rule

$$\begin{aligned} \boldsymbol{\delta}_j^{(l)} &= \frac{\partial C_{\text{in}}}{\partial \boldsymbol{x}_j^{(l)}} \cdot \frac{\partial \boldsymbol{x}_j^{(l)}}{\partial \boldsymbol{s}_j^{(l)}} \\ &= \theta'(\boldsymbol{s}_j^{(l)}) \cdot \frac{\partial C_{\text{in}}}{\partial \boldsymbol{x}_j^{(l)}} \end{aligned} \tag{88}$$

    where the unit output $\boldsymbol{x}_j^{(l)}$ is is given by (86)
- Recursive computation of of the sensitivity vector:
  - computing $\boldsymbol{\delta}^{(l)}$:
    * for last layer L, both terms in (88) are readily available
    * for the other layers, a recursion on the sensitivity vectors can be obtained by running, a slightly modified version of the neural network backwards

- since any component of $\boldsymbol{x}^{(l)}$ can affect $\boldsymbol{s}_k^{(l+1)}$, $\partial C_{\text{in}}/\partial \boldsymbol{x}_j^{(l)}$ in (88) is computed as follows

$$\begin{aligned} \frac{\partial C_{\text{in}}}{\partial \boldsymbol{x}_j^{(l)}} &= \sum_{k=1}^{d^{(l+1)}} \frac{\partial \boldsymbol{s}_k^{(l+1)}}{\partial \boldsymbol{x}_j^{(l)}} \frac{\partial C_{\text{in}}}{\partial \boldsymbol{s}_k^{(l+1)}} \\ &= \sum_{k=1}^{d^{(l+1)}} w_{jk}^{(l+1)} \boldsymbol{\delta}_k^{(l+1)} \end{aligned} \tag{89}$$

- substituting (89) into (88)

$$\boxed{\boldsymbol{\delta}_j^{(l)} = \theta'(\boldsymbol{s}_j^{(l)}) \sum_{k=1}^{d^{(l+1)}} w_{jk}^{(l+1)} \boldsymbol{\delta}_k^{(l+1)}} \tag{90}$$

- in vector form

$$\boldsymbol{\delta}^{(l)} = \theta'(\boldsymbol{s}^{(l)}) \otimes [\boldsymbol{W}^{(l+1)} \boldsymbol{\delta}^{(l+1)}]_1^{d^{(l)}} \tag{91}$$

  where $\otimes$ denotes component-wise multiplication, known as *Hadamard product* or *Schur product*

- The backpropagation algorithm:
  - for each batch $\{(\boldsymbol{x}_n, y_n)\}$, and for all i, j and l, it is desired to efficiently compute

$$\nabla C_{\text{in}} : \frac{\partial C_{\text{in}}}{\partial w_{ij}^{(l)}} \tag{92}$$

  - *backpropagation* is an algorithm that efficient computes (92)
    * backpropagation is usually applied using the batch gradient descent update rule
  - to express (92) in terms of sensitivity vector $\boldsymbol{\delta}^{(l)}$, first apply the chain rule, then substitute the components with (86, 87)

$$\begin{aligned} \frac{\partial C_{\text{in}}}{\partial w_{ij}^{(l)}} &= \frac{\partial C_{\text{in}}}{\partial \boldsymbol{s}_j^{(l)}} \cdot \frac{\partial \boldsymbol{s}_j^{(l)}}{\partial w_{ij}^{(l)}} \\ &= \boldsymbol{\delta}_j^{(l)} \cdot \boldsymbol{x}_i^{(l-1)} \Rightarrow \\ \frac{\partial C_{\text{in}}}{\partial \boldsymbol{W}^{(l)}} &= \boldsymbol{x}^{(l-1)} \cdot \boldsymbol{\delta}^{(l)} \end{aligned} \tag{93}$$

    * recursively computation of $\boldsymbol{\delta}^{(l)}$ (90), enables the backpropagation algorithm
  - in forward propagation, the nonlinearity was the activation $\theta(.)$, whereas in backpropagation, it is multiplication by $\theta'(\boldsymbol{s}_l^{(l)})$
  - the backpropagation algorithm described above is SGD based, where the weights change after processing each data sample $(\boldsymbol{x}_i, y_i)$
  - in practice, multiple $\boldsymbol{\delta}$s are computed before making a weight update
  - the size of the samples grouped together is called the *mini batch size*
  - more specifically, if the mini-batch size is m, then from (93)

$$\boldsymbol{W}^{(l)} \rightarrow \boldsymbol{W}^{(l)} - \frac{\eta}{m} \sum_x \boldsymbol{x}^{(l-1)} (\boldsymbol{\delta}_x^{(l)})^T \tag{94}$$

B. Hyper-parameters

- General remarks:
  - *hyper-parameters* are variables set before actually optimizing the weights of the NN
  - finding a pseudo-optimal set of such parameters is a major challenge
  - randomly generated parameters do in general better than grid search
- An *epoch* is a single forward and backward pass of the whole dataset
- List of *hyper-parameters* include:
  1) hidden layers, & nodes per layer
  2) cost function
  3) activation
  4) epochs & mini batch sizes
  5) learning rate

6) generalization parameter
7) weight initialization

- Weight initialization:
  - initialization is critical for deeper networks
  - if initial weights are large then many nodes will saturate
  - if initial weights are small then many node outputs stay close to zero, and weights do not get updated
  - with fan-in d, it is recommended to generate weights using
$$\mathcal{N}\left(0, \sqrt{\frac{\alpha}{d}}\right) \tag{95}$$
    where α is a hyper-parameter that is usually set to $\alpha = 1$ for $\tanh$ and $\alpha = 2$ for ReLU neuron models
  - bias is not sensitive to initialization, so can be set to 0, to Gaussian with standard deviation of one, or as in (95)

C. *Visualization*

- General remarks:
  - NN have high dimensionality and are hard to visualize
  - in *dimensionality reduction* high-dimensional data is mapped into lower dimensional data
    - ∗ one such method is PCA, see Appendix XV
    - ∗ another approach is to find maps that are distance preserving
  - visualization addresses the problem of visualizing high dimensional data
  - the space traversed by $\{\boldsymbol{x}_n\}$, at the input if a NN, is a very small subset of $\mathcal{R}^d$
- Multidimensional scaling
  - *multidimensional scaling*, or MDS, finds a distance preserving map by minimizing the cost function
$$\sum_{i,j}(d^*_{i,j} - d_{i,j})^2 \tag{96}$$
    where $d^*_{i,j}$, and $d_{i,j}$ are the distances between $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ in the original and reduced spaces respectively
- Sammon's mapping:
  - in Sammons mapping, more emphasis is given to preserving the distances between nearby points than between those which are far apart
  - the associated cost function is a modification of (96)
$$\sum_{i,j}\frac{(d^*_{i,j} - d_{i,j})^2}{d^*_{i,j}} \tag{97}$$
- Graph based visualization:
  - similar to Sammon's mapping, graph based visualization prioritizes local or nearby points
  - in a *nearest neighbor graph*, input vectors are the nodes and each node is connected to the three points that are closest to it in the original space
  - the associated cost function is
$$\sum_{i,j}\frac{1}{d_{i,j}} + \frac{1}{2}\sum_{(i,j)\in E}(d_{i,j} - d^*_{i,j})^2 \tag{98}$$
- t-SNE:
  - t-SNE stands for *t-distributed stochastic neighbor embedding*
  - t-SNE finds an embedding in two dimensions, such that locally, pairwise distances are conserved
  - t-SNE tries preserve the topology of the data
    - ∗ for every point, t-SNE constructs a notion of which other points are its neighbors, trying to make all points have the same number of neighbors
    - ∗ then it tries to embed them so that those points all have the same number of neighbors
    - ∗ t-SNE is similar to the graph based visualization, but instead of just having points be neighbors or not neighbors, t-SNE has a continuous spectrum of having points be neighbors to different extents
- Deconv approach to visualization:
  - to visualize some neuron, forward propagate signals to that neuron
  - set that neuron gradient to one and set all other neuron gradients in that layer to zero
  - back propagate and reconstruct the input

  - in (90), set

$$S_j^{(l)} \triangleq \sum_{k=1}^{d^{(l)}} w_{jk}^{(l)} \boldsymbol{\delta}_k^{(l)} \tag{99}$$

  - with *guided backpropagation*, backpropagation (90), is modified to

$$\boldsymbol{\delta}_j^{(l)} = \theta'(\boldsymbol{s}^{(l)})\,(S_j^{(l+1)} > 0)\,S_j^{(l+1)} \tag{100}$$

    i.e. only positive gradients are passed on to the previous layer, and so only neurons that positively help activation are incorporated
  - with *backward deconvnet*

$$\boldsymbol{\delta}_j^{(l)} = (S_j^{(l+1)} > 0)\,S_j^{(l+1)} \tag{101}$$

- Examples of visualization:
  - at first, or lowest, layer can use weights for visualization
    * the filters look like Gabor filters, see Section XV
  - at the top layers, check what set of stimuli does a neuron responds to
  - in an *occlusion experiment* the classification decision is monitored as a squared section of the input is zeroed out and moved around
- Optimization over image approaches:
  - the goal is to find an image that maximizes some score,
  - start by feeding all zeros
  - set the gradient of the scores vector to be $[0, 0, ...1, ..., 0]$, then back-propagate to image
  - update image
    * can blur it, or zero pixels with small norm
  - forward propagate
  - keep iterating
- Adversarial examples:
  - can take an arbitrary image, can make targeted but minor changes to the input so that it will be classified to any other false object
  - presently this is a concern

D. Deep learning

- Deep neural networks & deep learning:
  - the quintessential example of *deep neural networks* (DNN) is the feedforward ANN
    * DNN includes more hidden layers, and each hidden layer in general includes more neurons
  - in recent years, *deep learning* (DL) has become the most popular approach to developing AI
- Shallow versus deep NNs:
  - even though any function could be computed using a shallow network, it may not be a good choice
  - with DNN, it is possible to construct a solution through multiple layers of abstraction
  - for some functions very shallow circuits require exponentially more elements to compute a problem than do DNNs
- Unstable gradient as main challenge to DNNs:
  - in general, the gradient in DNNs is unstable, tending to either explode or vanish in earlier layers
    * this is due to the fact that the updated quantity is the product of terms from all the later layers
  - when the gradient tends to get smaller as we move backward through the hidden layers, then neurons in the earlier layers learn slower than neurons in later layers
    * this is known as the *vanishing gradient problem*
    * it occurs due to product of small quantities
  - in other instances, the gradient gets larger in earlier layers
    * this is known as *the exploding gradient problem*
    * it occurs if the weights in the product are of large quantities
- The traditional ML approach
  - to address unstable gradient problem, traditional ML approach was based on first developing hand-tuned feature extractors
  - kernel methods, Appendix XIII, can handle non-linearities but do not generalize well in practice

  - hand tuned feature extractors are not robust
  - it is difficult to design reliable feature extractors
- Deep belief networks:
  - *deep belief networks*, or DBNs, are a class of DNN that use RBMs (125), to train the initial layers
  - they were the first generation of DNNs to address unstable gradient problem
- Hierarchical approach in DL:
  - DL methods are representation-learning methods with multiple levels of representation, obtained by composing simple but non-linear modules that each transform the representation at one level (starting with the raw input) into a representation at a higher, slightly more abstract level
  - deep layers allow for hierarchical feature representations
  - with the composition of enough such transformations, very complex functions can be learned
  - each module in the stack transforms its input to increase both the selectivity and the invariance of the representation
  - for classification tasks, higher layers of representation amplify aspects of the input that are important for discrimination and suppress irrelevant variations
- Enablers of DNN:
  1) Diverse data availability
     - availability of digitized data due to internet, cloud etc.
     - techniques to artificially expand the training data, see Section I-B
  2) Improvements in learning algorithms
     - introduction of new activations such as ReLU (40)
     - regularization algorithms such as dropout
     - automatic feature extractors using convolutional & pooling layers
  3) Hardware acceleration
     - Moore's law
     - Graphic Processing Units (GPU), see Section X

E. *Convolutional, inception & residual networks*

- Convolutional neural network:
  - a *convolutional neural network*, or CNN, is a class within DNN
  - the *Neocognitron* (Fukushima, 1980) introduced a powerful model architecture for processing images, inspired by the structure of the mammalian visual system
  - Neocognitron is the basis for CNN network as proposed by LeCun et al., in 1998 used in vision tasks
- Network size concerns & remedies:
  - if size of data set is fixed, larger network usually means more parameters, making it prone to overfitting
  - larger networks increase computational resources
  - to remedy these two issues, additional constraints are introduced into the network such as regularization, sparsity, or structure
  - sparsity replaces the fully connected layers by sparser ones
    - todays computing infrastructures are inefficient when it comes to numerical calculation on non-uniform sparse data structures
  - convolutional neural networks introduce some form of constraint to the network
- CNNs use three ideas:
  1) Local receptive fields
  2) Shared weights
  3) Pooling
- Local receptive fields & stride length:
  - usually input data is represented as a rectangular format, rather than linear
  - in ANN there is full connectivity between layers $(l-1)$ and l
  - in a CNN the connections are in a small, localized regions of the input image
  - the region in the input image that is connected to a hidden neuron is called the *local receptive field* or a *patch*
  - different hidden neurons cover different local receptive fields
  - adjacent local receptive fields are separated by one or more pixels, known as the *stride length*
- Shared weights:
  - the set of hidden units that cover all the receptive fields, all share the same weights and biases

    * the *shared weights* and *bias* are often said to define a *kernel* or *filter bank*
  - every element of this set is extracting the same feature, but at a different position
    * the map from the input layer to the hidden layer is sometimes called a *feature map*
  - the weights and biases learned for a given output layer are shared across all patches in a given input layer
- Plurality of features:
  - in many visual applications, data at each layer is presented in 3-dimensions $(H \times W \times F)$
    * the amount of filters F, in a convolutional layer is called the *filter depth*
    * for input layer, the feature set could be the three primary colors
    * for hidden layers, multiple feature maps are usually extracted using multiple filter banks
  - consider the convolutional layer connection

$$(H \times W \times F_1) \xrightarrow{F_2 \times (3 \times 3 \times F_1)} (H \times W \times F_2) \tag{102}$$

    where each filter has $3 \times 3 \times F_1$ inputs, & for each location on the output layer there are F_2 filters
  - F_1 & F_2 refer to the feature sets at first and second layers
- Convolutional layer:
  - a hidden layer with shared filter bank is called a *convolutional* layer since the output is the convolution of a rectangular window and the input
  - it is common to zero pad the border if hidden layer sizes need to be preserved,
  - convolutional filters
    * provides translation invariance of images
    * dramatically reduces the number of parameters, compared to full FC layers
- On filter size trends: (1×1) convolutions
  - recently, multiple layers of smaller sized filters are chosen over a single layer of larger sized filters
    * such architectures, in general, use less memory, are more computationally efficient, & provide more non-linearities
  - can factor $(n \times n)$ convolutions into $(1 \times n)$ & $(n \times 1)$
  - in the extreme, can have $(1 \times 1 \times F)$ layers
- Example on filter size tradeoffs: a bottleneck architecture
  - consider the connection architecture of (102) with $F_1 = F_2 = F$
  - this connection architecture can be replaced by a *bottleneck architecture*, where

$$(H \times W \times F) \xrightarrow{\frac{F}{2} \times (1,1,F)} (H \times W \times \tfrac{F}{2}) \xrightarrow{F/2 \times (3,3,\frac{F}{2})} (H \times W \times \tfrac{F}{2}) \xrightarrow{F \times (1,1,\frac{F}{2})} (H \times W \times F) \tag{103}$$

    * first, the layer size is reduced to $(H \times W \times F/2)$ by using $F/2$ $(1 \times 1 \times F)$ filters per position
    * then another layer of same size is generated by using $F/2$ $(3 \times 3 \times F/2)$ filters
    * finally, original layer size $(H \times W \times F)$ is restored using F $(1 \times 1 \times F/2)$ filters
- Filter implementation:
  - *im2col*: convert both the input and weights into matrices, so that a layer update reduces to matrix multiplication
  - alternatively, for large filters, can use FFT
- Super-parameters needed for convolutional layer:
  1) Features or filer banks
     - choose a power of 2, like 64
  2) Filter size, or local receptive fields
     - use smaller sizes at lower layers and larger sizes for higher layers
  3) The stride length
     - choose 1 or 2
  4) Zero padding
- Pooling layer:
  - a *pooling layer* is usually used after convolutional layers
  - a pooling layer simplifies, or condenses, the output information from the convolutional layer
    * the role of the pooling layer is to merge semantically similar features into one
    * each feature map is pooled separately
  - conceptually, the benefit of the max pooling operation is to reduce the size of the input, and allow the neural network to focus on only the most important elements

- pooling reduces the dimension of the representation and creates an invariance to small shifts and distortions
  * pooling reduces the number of parameters needed in later layers
- common pooling strategies do not introduce new parameters but do introduce new hyper-parameters, including *pooling size* and *pooling stride*
- the pooling operation is applied individually for each depth slice
  * for a pooling layer the output depth is the same as the input depth
- (2×2) pooling with stride of 2 is common

• Pooling methods:
  - with *max-pooling*, a unit simply outputs the maximum activation in an input region

$$y = \max_i(X_i) \tag{104}$$

  - with *average pooling*, the average signal is picked

$$y = \text{mean}_i(X_i) \tag{105}$$

  - with *L2 pooling*, the square root of the sum of the squares of the activations are computed

$$y = \sqrt{\sum_i X_i^2} \tag{106}$$

• Hierarchical approach to the softmax output layer:
  - when there are large number of classes, the softmax output layer becomes complex
  - a hierarchical approach introduces classes c_t, where multiple classes would belong to class c_t
  - the conditional probably distribution can then be factored as

$$p(o_t|\text{history}) = p(c_t|\text{history})\, p(o_t|c_t) \tag{107}$$

• Inception network:
  - *inception* is computationally efficient architecture
  - inception-v1, see [3]:
    * an inception module or layer, is constructed by the concatenation of various filters such as pooling operations & convolutional filters of different width
    * an inception layer also uses filter size reduction as discussed in (102)
    * an *inception network* is a network consisting of Inception modules, stacked upon each other, with occasional max-pooling layers with stride 2 to halve the resolution of the grid
  - inception-v2 introduced batch normalization (12)
  - inception-v4 includes more diverse types of inception & reduction layers [4]

• Residual connections:
  - introduced by He et al. in 2015
  - residual connection utilize additive merging of signals from different layers
  - if x is the ReLU output from previous layer, then residual connection computes the next layer output as

$$y = \text{ReLU}[\text{Conv}(\text{Conv}(x)) + x] \tag{108}$$

• Trends:
  - smaller filters
  - deeper architectures
  - possibly dropping pooling
    * recent datasets are so big and complex we are more concerned about under-fitting
    * dropout is a better regularizer
    * pooling results in a loss of information
  - dropping some or all FC layers
  - special layers such as residual connections, inception layers or some combinations of the two

F. Transfer Learning

- *Transfer learning* involves taking a pre-trained neural network and re-adapting it to a new, different data set
- Approaches to transfer learning:
  1) new data set is small, new data is similar to original training data
     - retrain only top FC layer
  2) new data set is small, new data is different from original training data
     - replace all FC layers and last convolutional layer with a single FC layer
     - this is known as *feature extraction*
  3) new data set is large, new data is similar to original training data
     - remove the last FC layer and replace with a layer matching the number of classes in the new data set
     - initialize the rest of the weights using the pre-trained weights
     - re-train the entire neural network
     - this is known as *finetuning*
  4) new data set is large, new data is different from original training data
     - remove the last FC layer and replace with a layer matching the number of classes in the new data set
     - retrain the network from scratch with randomly initialized weights
- With modest training data, can retrain both middle and upper layers
  - re-learn middle layers with 1% learning rate of original network
  - re-learn upper layers with 10% learning rate of original network
- Popular networks to start from:
  1) LeNet [5]
     - the first successful applications of CNN, in 1998
     - best known for reading zip codes, digits, etc.
  2) AlexNet [6]
     - popularized CNN in computer vision by winning ImageNet 2012, by a significant margin
     - compared to LeNet, its deeper, bigger, & featured convolutional layers stacked on top of each other
     - used ReLU activation and dropout to avoid overfitting
     - 8 layers, 5 convolution layers followed by 3 FC layers
  3) ZFNet [7]
     - winner of ImageNet 2013
     - compared to AlexNet, the size of the middle convolutional layers were expanded and the stride and filter size on the first layer were reduced
  4) GoogLeNet [3]
     - winner of ImageNet 2014
     - the *GoogLeNet* name refers to a particular incarnation of the inception architecture used in the submission for the ILSVRC 2014 competition
     - it is known to be fast and has potential for real-time applications
     - 22 layers with parameters, i.e. without pooling
  5) VGGNet [8]
     - from Visual Geometry Group at Oxford University
     - the runner-up in ImageNet 2014
     - it features a homogeneous architecture that only performs (3×3) convolutions with (2×2) pooling layers, followed by three layers of FC layers
     - known to be a flexible network
     - but it is expensive to evaluate and uses a lot more memory and parameters (140M)
     - 19 total layers
  6) ResNet [9]
     - winner of ImageNet 2015
     - *ResNet*, short for residual network
     - ResNet is an ultra-deep network with 152 layers
     - ResNet uses residual connections (108), batch normalization (12), and (1×1) convolutional filters (102)
       * ResNet does not include FC fully layers at the top of the network
     - ResNet is currently the state of the art of CNNs, and are the default choice for using CNNs in practice

V. Recurrent Neural Networks with Supervision

- Overview:
  - *Recurrent Neural Networks* (RNN) entail feedback,
  - RNN networks are effective for
    - ∗ visual attention systems work sequentially on an input,
    - ∗ systems that receive a sequence of inputs, such as sentiment analysis,
    - ∗ systems that produce a sequence of outputs, for example, captioning an image,
    - ∗ systems with both sequential inputs and outputs such as machine translation,
  - RNNs can be viewed as programs, or state machine,
  - the final state vector of an RNN's hidden units may represent the thought expressed by the sentence.
- RNN model:
  - if $\boldsymbol{x}$ is the input, and $\boldsymbol{h}$ is output of the hidden layer, then

$$\boldsymbol{h}_t = f\left(\boldsymbol{x}_t, \boldsymbol{h}_{t-1}\right), \tag{109}$$

  - if hidden layers form a single layer,

$$\boldsymbol{h}_t = \theta\left(W^{(x)}\boldsymbol{x}_t + W\,\boldsymbol{h}_{t-1}\right), \tag{110}$$

  - alternatively, if $\boldsymbol{h}$ is the signal level at the hidden layer, then the an RNN can be modeled as

$$\boldsymbol{h}_t = W^{(x)}\boldsymbol{x}_t + W\,\theta(\boldsymbol{h}_{t-1}), \tag{111}$$

  - then the output is of the form

$$\hat{y}_t = W^{(s)}\,\theta(\boldsymbol{h}_t). \tag{112}$$

- Deep RNN:
  - stack multiple RNN layers, where the hidden state of one RNN is the input of the next RNN,
  - there are multiple hidden layers between inputs and outputs.
- Bidirectional RNN:
  - causal is forward, anti-causal in backwards, like BCJR,
  - the output is a function of the concatenation of the two hidden layer outputs.
- Deep bidirectional RNN:
  - a deep and bidirectional extensions of an RNN can be combined to create a deep bidirectional RNN.
- Encoder and decoders:
  - for some applications, such and language translation, can think of the system model of doing two tasks, encoding and decoding
  - in the basic RNN model (111)
    - ∗ encoding occurs while there is an incoming stream of inputs $\boldsymbol{x}_t$,
    - ∗ decoding occurs when there is an output $\hat{y}_t$,
  - assume decoder starts after encoder ends.
- Extensions to RNN for encoders/decoders:
  - this basic model (111), can be extended in multiple ways:
    - ∗ use different weights W', for decoding,

$$\boldsymbol{h}_{D,t} = W'\,\theta(\boldsymbol{h}_{t-1}), \tag{113}$$

    - ∗ remember last state of the encoder $\boldsymbol{h}_T$,

$$\boldsymbol{h}_{D,t} = W'\,\theta(\boldsymbol{h}_{t-1}) + W^T\theta(\boldsymbol{h}_T) \tag{114}$$

    - ∗ include the previous output $\hat{y}_{t-1}$,

$$\boldsymbol{h}_{D,t} = W'\,\theta(\boldsymbol{h}_{t-1}) + W^T\theta(\boldsymbol{h}_T) + W^y\hat{y}_{t-1}. \tag{115}$$

- Vanishing / exploding problem in RNN:
  - denote the cost function at time t as E_t,

- the derivative of E_t w.r.t. W can be expressed as,

$$\frac{\partial E_t}{\partial W} = \sum_{\tau=1}^{t} \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial \boldsymbol{h}_t} \frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_\tau} \frac{\partial \boldsymbol{h}_\tau}{\partial W} \tag{116}$$

- using (111), expand one of Jacobian terms in (116),

$$\begin{aligned} \frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_\tau} &= \prod_{j=\tau+1}^{t} \frac{\partial \boldsymbol{h}_j}{\partial \boldsymbol{h}_{j-1}} \\ &= \prod_{j=\tau+1}^{t} W^T \text{diag}[\theta'(\boldsymbol{h}_{j-1})], \end{aligned} \tag{117}$$

- then the gradient becomes the product of all these Jacobian matrices which becomes very small or very large quickly,
- for exploding problem can saturate the gradient,
- for vanishing problem initialize $W = I$ and use ReLU.

• Learning with long term memory:
  - theoretical and empirical evidence shows that it is difficult to learn to store information for long sequences,
  - three approaches include,
    * gating techniques,
    * *Neural Turing Machine*, where the network is augmented by a tape-like memory that the RNN can choose to read from or write to,
    * *memory networks*, where a regular network is augmented by a kind of associative memory.

• Gating:
  - *gating* is a technique that helps the network to decide when to forget or remember an input,
  - gating techniques use RNN where each hidden layer is more complex,
  - two popular gating techniques are GRU and LSTM,

• GRU:
  - the ideas of *Gated Recurrent Units* (GRU) are,
    * to keeps around memories to capture long distance dependencies,
    * allow error messages to flow at different strengths depending on inputs,
  - the GRU first computes two gates, *update gate* z_t, and the *reset gate* r_t,

$$\begin{aligned} z_t &\triangleq \sigma\left(U^{(z)}x_t + W^{(z)}h_{t-1}\right), \\ r_t &\triangleq \sigma\left(U^{(r)}x_t + W^{(r)}h_{t-1}\right), \end{aligned} \tag{118}$$

    * σ is the sigmoid activation (34),
    * h_t is the output of the activation,
    * think of these two gates as a special hidden layers,
  - intermediate memory content is

$$\tilde{h}_t \triangleq \tanh(Wx_t + r_t \circ Uh_{t-1}), \tag{119}$$

  where $\circ$ is the Hadamard multiplication,
  - the hidden layer is updated as

$$h_t \triangleq z_t \circ h_{t-1} + (1 - z_t) \circ \tilde{h}_t. \tag{120}$$

• LSTM:
  - *Long short-term memory* (LSTM) [10], is a popular and powerful tool,
  - LSTM is similar to GRU but three gates are used rather than two: input gate i_t, forget gate f_t, and output gate o_t,

$$\begin{aligned} i_t &\triangleq \sigma\left(W^{(i)}x_t + U^{(i)}h_{t-1}\right), \\ f_t &\triangleq \sigma\left(W^{(f)}x_t + U^{(f)}h_{t-1}\right), \\ o_t &\triangleq \sigma\left(W^{(o)}x_t + U^{(o)}h_{t-1}\right), \end{aligned} \tag{121}$$

  - the intermediate memory cell $\tilde{c}_t$, the final memory cell c_t, and the final hidden state h_t, are respectively,

$$\begin{aligned} \tilde{c}_t &\triangleq \tanh\left(W^{(c)}x_t + U^{(c)}h_{t-1}\right), \\ c_t &\triangleq f_t \circ c_{t-1} + i_t \circ \tilde{c}_t, \\ h_t &\triangleq o_t \circ \tanh(c_t), \end{aligned} \tag{122}$$

  - c_t introduces additive interaction, similar to a ResNet,
  - these additions also help backpropagation,
  - multiple LSTM's are usually stacked up to generate a deep LSTM.
- Recursive neural networks are covered in Section IX.

VI. Unsupervised Learning

- Unsupervised learning:
  - the goal of *unsupervised learning* is to learn some structure of the data, without labels or teacher,
  - still mostly a research topic,
  - examples include
    - clustering,
      - k-means clustering is unsupervised clustering algorithm, not based in NN, see Appendix XV.
    - dimensionality reduction & feature learning,
      - see autoencoders below,
    - generative modeling,
      - see variational autoencoders below.
- Hebbian rule:
  - neurons that fire together, wire together,
  - more specifically, *Hebbian rule*, is a simple, unsupervised learning algorithm [11], where the weight w_{ij} connecting neurons x_i to x_j is updated proportional to

$$\frac{dw_{ij}}{dt} \propto \text{correlation}(x_i x_j). \tag{123}$$

- Energy-based models:
  - *energy-based models* associate a scalar energy to each configuration of the variables of interest,
  - learning corresponds to modifying the energy function so that its shape has desirable properties,
    - for example, we would like desirable configurations to have low energy,
  - Hopfield networks and Boltzmann machines are examples of energy-based models.
- Hopfield networks:
  - a *Hopfield network* is an undirected graph with only visible nodes,
  - a Hopfield model is an energy-based model, where the goal is to make the N samples $\{\boldsymbol{x}_i\}$, the stable states of the network,
  - the neurons can be activated synchronously (simultaneously), or asynchronously (one at a time),
  - applications:
    - associative memories, or content addressable memories,
    - optimization problems.
- Hopfield network learning:
  - let $x_i^{(n)} \in [-1, 1]$, be the activation output,
  - the weights are set based on Hebbian learning (123),

$$w_{ij} \sim \sum_{n \leq N} x_i^{(n)} x_j^{(n)}, \tag{124}$$

  - the resulting learning algorithm is unsupervised,
  - the weights are symmetric, i.e., $w_{ij} = w_{ji}$,
  - there are no self connections, i.e., set $w_{ii} = 0$.
- Boltzmann machines & RBM:
  - a *Boltzmann machine* differs from a Hopfield network in two ways,
    - the Boltzmann machine also allows hidden nodes,
    - the Boltzmann machine uses stochastic neuron, with probabilistic firing mechanism,
  - a *restricted Boltzmann machine* (RBM) is a Boltzmann machine with a bipartite structure

$$\boldsymbol{x} \leftrightarrow \boldsymbol{h} \tag{125}$$

  where the hidden nodes constitute one layer and visible nodes the other,
    - there are no connections within a layer.

- Autoencoders:
  - an *autoencoder* is trained to encode the input $\boldsymbol{x}$ into some representation $c(\boldsymbol{x})$, so that $\boldsymbol{x}$ can be reconstructed from $c(\boldsymbol{x})$,
$$\boldsymbol{x} \xrightarrow{\text{encoder}} c(\boldsymbol{x}) \xrightarrow{\text{decoder}} \boldsymbol{x} \tag{126}$$
    - autoencoders are symmetric in the sense that the encoder and a decoder could share weights,
    - the encoder could be a deep network,
    - $c(\boldsymbol{x})$ is viewed as a lossy compression of $\boldsymbol{x}$,
    - can think of $c(\boldsymbol{x})$ as features of $\boldsymbol{x}$,
  - the associated cost function is
$$-\log p(\boldsymbol{x}|c(\boldsymbol{x})), \tag{127}$$
  - as an example, if there is one linear hidden layer and the mean squared error criterion is used to train the network, then the k hidden units learn to project the input in the span of the first k principal components of the data,
  - an autoencoder is usually used as component of neural network, such as feature extraction, and dimensionality reduction, where only the encoder is used after training,
    - autoencoders are not very effective in training deep networks.
- Autoencoder versus RBM:
  - taking $\boldsymbol{h} = c(\boldsymbol{x})$ in (125, 126), an autoencoder and RBM become equivalent when,
    - the encoder is one layer,
    - the encoder and decoder are symmetric and share weights,
    - an RBM uses the same activation as an autoencoder.
- Variation autoencoder:
  - using a Bayesian approach on an autoencoder, a *variational autoencoder* (VAE) can generate new data,
    - generating new data from available one is called generative modeling,
  - in VAE, $\boldsymbol{z} \triangleq c(\boldsymbol{x})$ (see (126)) is called the *latent variables*,
    - think of $\boldsymbol{z}_i$ as an object or a high level attribute of an object,
    - $P(\boldsymbol{z})$ is assumed to be diagonal, unit Gaussian random vector,
  - assume $P(\boldsymbol{x}|\boldsymbol{z})$ & $P(\boldsymbol{z}|\boldsymbol{x})$ are also diagonal Gaussian, with mean and variances of (μ, Σ),
  - the encoder & decoder are used to predict the corresponding means & variances,
$$\begin{aligned} \boldsymbol{z} \in P(\boldsymbol{z}) \quad & \xrightarrow{\text{VAE decoder}} (\mu^x, \Sigma^x), \\ \boldsymbol{x} \quad & \xrightarrow{\text{VAE encoder}} (\mu^z, \Sigma^z), \end{aligned} \tag{128}$$
  - during training the process is as follows
$$\boldsymbol{x} \to (\mu^z, \Sigma^z) \to \boldsymbol{z} \to (\mu^x, \Sigma^x) \to \boldsymbol{x}', \tag{129}$$
  - the loss function is the *reconstruction loss* which could be based on KL divergence rather than L2,
  - to generate data,
$$\boldsymbol{z} \to (\mu^x, \Sigma^x) \to \boldsymbol{x}. \tag{130}$$
  - Generative adversarial networks:
    - *generative adversarial networks* is a framework for estimating generative models via an adversarial process, with two components
      - a generative model (G), similar to VAE decoder, generates fake data,
      - a discriminator model (D), is a binary classifier that decides whether a data is real or generated by G.
    - generator and discriminator model are trained together,

VII. Applications

- Andrew Ng: AI is the new electricity
  - rule of thumb: what humans can do in less than a second
- Applications:
  - vision, Section VIII
  - automatic speech recognition & natural language processing, Section IX
  - health care & biomedicine, Section VII-A
  - web searches, content filtering & online advertising

        - what ad to provide to a given user?
    - personal assistant such as e-mail smart reply
    - recommendations on e-commerce web sites
        - Netflix
    - automotive & robotics
    - analyzing particle accelerator data
    - video games
    - finance
    - education
- Search engine:
    - given a query, a search engines can also search for other phrases that have word vectors close to the query, see Section IX,
    - given a query and a document, a neural network can return a score for the query, document pair.
- Some startups:
    - Capio, clarifai, calrify, Dato, emotient, enlitic, ersatz labs, EyeEm, herta security, IFLYTEK, Intelligent Voice, iQIY, Letv, megv11, MetaMind, NERVANA SYSTEMS, rbeus, SENSETIME, Sogou, Unisound, VIONVISION, zebra.

A. Health care & Biomedicine

- Medical image understanding
    - radiology
- Massive amount of genomic data availability
- Bioinformatics:
    - can think of gene expression as object classification
    - drug discovery
- Molecular activity prediction

VIII. Visual Recognition

- Generalities:
  - vision recognition is also known as *image perception*, *machine vision*, or *computer vision*
  - computer vision enables computers to see, identify and process images in the same way that human vision does, and then provide appropriate output
    - the goal of computer vision is to extract useful information from an image
  - a camera looks at a 3D object and transforms them to 2D image
    - this transformation is modeled by a *camera matrix* C
    - computer vision can infer depth resolution, i.e. 3D
  - compared to other sensors
    - images are characterized as having high spatial resolution
    - cameras and image processing are cheap
- A list of vision tasks:
  - classification (of a single object)
    - *classification* implies single object classification
    - classification partitions images to C classes, or labels
    - e.g., face recognition, identity verification
  - localization (put box around an object), see below
  - classification + localization
    - generates both the label as well as the location of an object
  - detection (classification + localization of variable number of objects)
  - semantic segmentation (classification at pixel level)
  - instance segmentation
  - attention models
  - video & action recognition
  - 3D and depth interpretation
- Localization:
  - *localization* implies localizing a classified object
  - given an image, generate box around object
    - box means object coordinates at upper left corner (x, y), width and height (w, h)
    - sometimes referred to as *bounding-box regression*
  - can be approached as a regression problem, where the metric is the L2 distance between estimated box and ideal box coordinates
  - metric can also be generated as the intersection of box and object, divided by their union
- Localization of K objects:
  - K is the number of objects in an image
  - for example, $K = 4$

$$\{\text{cat, cat's head, left ear, right ear}\} \tag{131}$$

  - in the above example, the output is $4K$ values
- Image location to other regression tasks:
  - the localization regression technique can also be used to characterize other real valued properties of an object
  - for example, human pose estimation
- Detection & mAP:
  - *detection* is the classification and the localization of variable number of objects,
  - evaluation is through the metric *mean average precision*, or mAP
    - mAP computes average precision separately for each class, then averages over classes,
    - $0 \leq \text{mAP} \leq 100$, high is good

A. Color

- Color space:
  - a *color space* is a specific organization of colors
  - a color space provides a way to categorize colors and represent them in digital images
  - the color space of a pixel is usually represented by a triplet while greyscale is a scalar

- Hue:
  - hue represents color independent of any change in brightness
  - *hue* is in degrees on the color wheel
    - hue is a cylindrical-coordinate representation
  - let $V_{\text{max}} \triangleq \max(R, G, B)$, $V_{\text{min}} \triangleq \min(R, G, B)$ and $\Delta = V_{\text{max}} - V_{\text{min}}$, then the hue H is

$$\begin{aligned} H &= \frac{30}{\Delta}(G - B), && \text{if } V_{\text{max}} = R \\ H &= 60 + \frac{30}{\Delta}(B - R), && \text{if } V_{\text{max}} = G \\ H &= 120 + \frac{30}{\Delta}(R - G), && \text{if } V_{\text{max}} = B \end{aligned} \tag{132}$$

    - 0 (or $360/2$) is red, $120/2$ is green, $240/2$ is blue
- Lightness or value:
  - *lightness* measures the relative lightness or darkness of a color
  - lightness is a percentage value
    - 0% is dark (black)
    - 100% is light (white)
  - if $V_{\text{max}} \triangleq \max(R, G, B)$, and $V_{\text{min}} \triangleq \min(R, G, B)$, then the lightness L is

$$L = \frac{V_{\text{max}} + V_{\text{min}}}{2} \tag{133}$$

- Saturation:
  - *saturation* is a measurement of colorfulness
    - as colors get lighter and closer to white, they have a lower saturation value, whereas colors that are the most intense, like a bright primary color have a high saturation value
  - saturation is a percentage value
    - 100% is the full colour
  - if $V_{\text{max}} \triangleq \max(R, G, B)$, and $V_{\text{min}} \triangleq \min(R, G, B)$, then the saturation S is

$$\begin{aligned} S &= \frac{V_{\text{max}} - V_{\text{min}}}{V_{\text{max}} + V_{\text{min}}} && \text{if } L < 0.5 \\ S &= \frac{V_{\text{max}} - V_{\text{min}}}{2 - (V_{\text{max}} + V_{\text{min}})} && \text{if } L \geq 0.5 \end{aligned} \tag{134}$$

- Color space representations:
  - RGB (Red, Green, Blue)
  - BGR (Blue, Green, Red)
  - HSV (Hue, Saturation, Value)
  - HLS (Hue, Lightness, Saturation)
  - LUV (Luminance, while $(x, y) \rightarrow (u, v)$)
  - Lab (Lightness, a and b for the color opponents green-red and blue-yellow)

B. Dataset

- Data sets:
  - MNIST, see below
  - CIFAR, see below
  - ImageNet
    - *ImageNet Large Scale Visual Recognition Challenge*, or ILSVRC
    - a database of labeled images
  - vehicle image database, GTI, http://www.gti.ssr.upm.es/data/Vehicle_database.html
  - KITTI, http://www.cvlibs.net/datasets/kitti/
  - PASCAL VOC
  - MS-COCO
- MNIST data set:
  - MNIST data set is a database of the 10 digits

  - http://yann.lecun.com/exdb/mnist/
  - NIST stands for the United States' *National Institute of Standards and Technology*
  - MNIST is a *modified* subset of two data sets collected by NIST
  - specifics:
    - the images are greyscale, (28×28) pixels
    - training data contains $60,000$ images
      - these images are scanned handwriting samples from 250 people, half of whom were US Census Bureau employees, and half of whom were high school students
    - test data contains $10,000$ images
      - the test data was taken from a different set of 250 people than the original training data (a group split between Census Bureau employees and high school students)
- CIFAR:
  - https://www.cs.toronto.edu/~kriz/cifar.html
  - the CIFAR-10 and CIFAR-100 are labeled subsets of the 80 million tiny images dataset
  - CIFAR-10 dataset:
    - $60,000$ images
      - $50,000$ training images and $10,000$ test images
    - 32×32 colour images
    - 10 classes
      - with 6000 images per class
  - CIFAR-100 dataset:
    - like the CIFAR-10, except it has 100 classes containing 600 images each
    - there are 500 training images and 100 testing images per class
- Formats:
  - LMDB
  - HDF5

C. Image Processing

- Edge detection:
  - an edge detection algorithm extracts edges from an image
  - an image is converted to a binary image where the ones correspond to the edges
  - approaches to detect edges
    - gradient based, for example by using Sobel operator or Canny Edge detection, where first gradient is taken and then passed through a threshold
    - threshold based, by looking into some of the color space attributes and then passing it through a threshold
- Image convolution:
  - an image $f(x, y)$ is convolved with a *filter*, or *mask* $h(x, y)$

$$g(x, y) = h(x, y) * f(x, y) \tag{135}$$

  - $h(x, y)$ is $(n \times n)$, when n is odd, e.g. $n = 3$
  - applications include blurring, sharpening, edge detection, noise reduction etc.
    - Gaussian smoothing
    - Sobel operator
- Sobel operator:
  - the Sobel operators or filters are defined as

$$S_x \triangleq \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}, \qquad S_y \triangleq \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} \tag{136}$$

  - the Sobel operators take the derivative of the image in the x (horizontal) or y (vertical) direction
- Canny Edge Detection:
  - developed by John F. Canny in 1986
  - the goal is to identify the boundaries in an image
  - the Sobel operator is at the heart of the Canny edge detection algorithm

- the following steps are usually included:
  - convert image to greyscale
  - apply Gaussian smoothing function to suppress noise and spurious gradients
  - compute the gradient
  - have two thresholds, where all edges detected above the high threshold are retained while pixels between two threshold are retained if connected to strong edges
- The Hough Transform & lines:
  - developed by Paul Hough in 1962
  - *Hough transform* represents lines in parameter space, or in *Hough space*
    - the line $y = mx + b$ (in *image space*) transforms to the point (m, b) in Hough space

$$(x, y) \rightarrow (m, b) \tag{137}$$

    - a point (x_o, y_o) transforms to the line $(m, y_o - mx_o)$ over variable m
      - a line is transformed to a point
      - a point is transformed to line
    - a line that passes through two points in image space is the intersection of the two lines, associated with the two points, in Hough space
  - instead of detecting lines, from multiple points, in image space, can detect intersection of lines in Hough space
- Hough Transform in polar coordinates:
  - polar coordinates (ρ, θ), are used to address vertical line representations with $m = \infty$,
  - draw a segment s, from origin that intersects perpendicular to $y = mx + b$
    - ρ is the perpendicular distance of s
    - θ is the angle at origin from x-axis to s
  - since the product of the slopes of perpendicular lines is -1

$$-1 = m \cdot \tan\theta \Rightarrow m = -\frac{\cos\theta}{\sin\theta} \tag{138}$$

  - using simple geometry

$$\sin\theta = \frac{\rho}{b} \Rightarrow b = \frac{\rho}{\sin\theta} \tag{139}$$

  - substituting m and b into the line equation

$$\begin{aligned} y &= mx + b \\ &= -\frac{\cos\theta}{\sin\theta}x + \frac{\rho}{\sin\theta} \Rightarrow \\ x\cos\theta + y\sin\theta &= \rho \end{aligned} \tag{140}$$

  - a point in image space (x_o, y_o) corresponds to the sinusoidal equation in Hough space

$$x_o \cos\theta + y_o \sin\theta = \rho \tag{141}$$

  - detecting lines (from points) in image space becomes equivalent to detecting intersection of sinusoidal equations in Hough space
- Distortion:
  - during 3D to 2D transformation of a camera capture, distortion changes the shape and size of 3D objects captured on 2D image
  - types of distortion:
    - radial distortion
    - tangential distortion
  - a regular pattern, like a chessboard, can be used to calibrate a camera
- Radial distortion:
  - cameras use curved lenses to form an image, and light rays often bend a little too much or too little at the edges of these lenses
  - this creates an effect that distorts the edges of images, where lines or objects appear more or less curved than they actually are
  - this phenomenon is known as *radial distortion*, and its the most common type of distortion
  - let (x, y) be a point on the distorted image

  - let r be the distance from *distortion center* to (x, y)
  - to correct the appearance of radially distorted points in an image, a correction formula can be applied that uses three distortion coefficients $k1$, $k2$, $k3$, and r

$$\begin{aligned} x_c &= x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \\ y_c &= y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \end{aligned} \tag{142}$$

- Tangential distortion:
  - tangential distortion occurs when a cameras lens is not aligned perfectly parallel to the imaging plane, where the camera film or sensor is
  - this makes an image look tilted so that some objects appear farther away or closer than they actually are
  - given the two distortion coefficients p_1 & $p2$, the correction formulae is

$$\begin{aligned} x_c &= x + 2p_1 xy + p_2(r^2 + 2x^2) \\ y_c &= y + 2p_2 xy + p_1(r^2 + 2y^2) \end{aligned} \tag{143}$$

- Perspective transform:
  - a perspective transform maps the points in a given image to different, desired, image points with a new perspective
  - the birds-eye view transform that lets us view a lane from above is useful for calculating the lane curvature

D. *Vision with legacy machine learning*

- Features:
  - raw pixel intensity
    - it can provide both color & shape characteristics about an object
    - for example, template matching
  - histogram of pixel intensity
    - it provides color characteristics about an object
    - histograms lose shape characteristics but make detection less sensitive to orientation
    - for example, saturation histogram forms a feature vector
  - gradient of pixel intensity
    - it provides shape characteristic of an object
    - it loses color characteristics & becomes color invariant
    - for example, the direction of gradient can be used to identify a shape, see HOG below
  - recommended to use multiple features
    - need to normalize features if all features are represented as a single vector
  - finally, the features are fed to a classifier to be trained on data
- HOG:
  - *histogram of oriented gradients* (HOG) is a detection algorithm from 2005
  - HOG assumes that an object can be described by the distribution (histogram) of intensity gradients (edge directions)
  - an image is divided into small connected regions called *cells*, and for the pixels within each cell, a histogram of gradient directions is compiled
    - orientation is quantized to M bins
    - each pixel in a cell is assigned to one of the M orientations
    - each pixel influences on that bin based on its magnitude
    - the resulting histogram is the feature vector associated with each cell
  - hyperparameters include cell size, cell overlap, quantized bins M,
  - afterwards a linear classifier is applied on these features
  - there are similarities to CNNs
- DPM:
  - *deformable parts model* (DPM), from 2010, is based on HOG but uses more more complex functions than linear classifier
  - DPM was state of the art before CNNs
- Sliding window & Overfeat:
  - so far feature extraction and classification was performed on a single window on a larger image
  - to be able to do detection, run classification at multiple locations & scales on a high resolution image
  - at each location generate score

  - using heuristics, combine classifier & regressor predictions across all scales for final prediction
- Overfeat:
  - Overfeat [12] is an example of a sliding window architecture
  - Overfeat was the winner of ILSVRC 2013 classification + localization challenge

E. *Vision with neural networks*

- Classification and localization with CNN:
  - the corresponding CNN network is the union of some CNN for classification, with a second FC *regression-head*, appended to the network, at the top
    - the resulting CNN will have two heads, a *classification-head* and a regression-head
    - the regression head gets its input either from the final convolutional layer, or after last FC layer
    - the regression head is trained separately
    - localization can be class agnostic, with 4 coordinate numbers, or class specific, with $4C$ numbers
- Detection as classification:
  - detection can be reduced to classification / localization problem by using moving window techniques
  - need to test many positions and scales
- Region proposal and selective search:
  - *region proposal* algorithm looks only at a small subset of possible positions
  - it finds blobby image regions that are likely to contain objects
  - it performs class-agnostic object detector on a region
  - one method of doing region proposal is called *selective search* which is a bottom-up segmentation approach , merging regions at multiple scales
- R-CNN [13]:
  - *region based CNN*, or R-CNN is a region based algorithm proposal, implemented on a CNN
  - using selective search module (not NN), called *region proposal*, get around 2000 different sized regions of interest (RoI)
  - crop and warp *each* region to some fixed size
  - run CNN on each region of interest
  - the classification-head used was SVM with capability of classifying $C = 21$ different objects
  - R-CNN is slow, with multistage training pipeline
- Fast R-CNN [14],
  - to minimize process time
$$\text{R-CNN} \rightarrow \text{Fast R-CNN} \tag{144}$$
  - in fast R-CNN, an image goes through CNN once, not per RoI
  - the flow is as follows
$$\begin{tikzcd} \boldsymbol{x} \arrow[r, "\text{CNN}"] & \text{feature map} \arrow[r, "\text{region proposal}"] & \text{RoIs} \arrow[d, "\text{1layer}"'] \\ & \text{classifier, loc.} & \text{pooled RoI} \arrow[l, "\text{FCs}"'] \end{tikzcd} \tag{145}$$
  - region proposal module is not NN
    - region proposal is the bottleneck in fast R-CNN
  - course localization in inferred from the RoI, fine localization from regression head
- RPN:
  - RPN stands for *region proposal network*
  - RPN takes an image as input and outputs a set of rectangular object proposals, each with a score
  - RPN is modeled as a fully deep convolutional network
  - RPN is trained to produce region proposals directly; no need for external region proposals
  - for each position N rectangular sizes are considered called *anchor boxes*
- Faster R-CNN [15]:
  - with *faster R-CNN*, region proposal is done by RPN
  - the entire system is a single, unified network for object detection
  - two level of decisions:
    - whether there is a RoI

    - ∗ what the object is
  - results were further improved by using faster R-CNN with ResNet [16]
- Image segmentation:
  - image segmentation aims to group perceptually similar pixels into regions and is a fundamental problem in computer vision
  - two segmentation types:
    1) semantic segmentation
    2) instance segmentation
- Semantic segmentation:
  - with *semantic segmentation*, every pixel in an image is classified to belong to some object
  - different instances of the same object are not differentiated
  - if different object types are labeled by different colors, then every pixel in an image is colored by a classification
- Semantic segmentation using NN:
  - semantic segmentation can be performed by extracting patches, performing patch classification, and by labeling the patch center-pixel by that object
    - ∗ this method is straight forward but tedious and expensive
  - similar to detection (R-CNN), can run CNN on the whole image once, and then pick patches from higher layers
  - *refinement* combines CNN and recurrence by using raw image and output of previous iteration as inputs for next iteration
- Upsampling:
  - *upsampling* addresses the issue of mapping CNN upper layers (with lower dimensions) to higher (pixel) dimensions
    - ∗ upsampling is the opposite of downsampling that occurs in generic CNNs
  - upsampling can be generated using by a union of inner layers using skipping
  - another approach is to use convolution type technique, where a filter will take a single input and will copy it to all $(n \times n)$ output values, each scaled by some filter weight w_{ij}
    - ∗ a stride length of > 1 is used, at the output rather than the input
    - ∗ overlapping outputs, from multiple filters, sum their values
  - sometimes referred to as *backward strided convolution*
- Instance segmentation:
  - *instance segmentation* is similar to semantic segmentation, however different instances of the same object are differentiated
  - sometimes called *simultaneous detection and segmentation* or SDS
  - can use combination of
    - ∗ region proposal network (RPN)
    - ∗ capturing RoI and passing them through CNN to generate segmentation masks
    - ∗ mask our background and predict object class
- Attention:
  - visual attention is the ability to focus on a certain region of an image with high resolution, while perceiving the surrounding image in low resolution
  - in NLP, see Section IX, attention is the ability to focus on certain words in a sentence
    - ∗ examples include translation, question & answer
- Visual Attention with captioning, a case study [17]:
  - attention can be used in image captioning, where the network flow is as follows

$$\begin{tikzcd}
 & & & {} \arrow[d] & {} \arrow[l] \\
\boldsymbol{x} \arrow[r, "\text{CNN}"] & a \arrow[r, "\phi"] \arrow[d] & \boldsymbol{z}_t \arrow[r, "\text{LSTM}"] & \boldsymbol{h}_{t-1} \arrow[r, "\text{NN}"] \arrow[d] & y_t \arrow[u] \\
 & {} \arrow[r] & \text{f} \arrow[u, "\alpha_i"] & {} \arrow[l] &
\end{tikzcd} \tag{146}$$

    - ∗ a is a set of L features, with $\boldsymbol{a}_i \in \mathbb{R}^d$
    - ∗ the *context vector* $\boldsymbol{z}_t$ is a dynamic representation of the relevant part of the image input at time t
    - ∗ $\boldsymbol{h}_t$ is the hidden state of an LSTM network

    * the caption is

$$y = \{\boldsymbol{y}_1, \cdots, \boldsymbol{y}_C\}, \quad \boldsymbol{y}_i \in \mathbb{R}^K \tag{147}$$

      where K is vocabulary size and C is caption length
    * f is the *attention model*, where ignoring normalization

$$\alpha_{ti} = f(\{\boldsymbol{a}_i\}, \boldsymbol{h}_{t-1}) \tag{148}$$

    * given the weights α_i

$$\boldsymbol{z}_t = \phi(\{\boldsymbol{a}_i\}, \{\alpha_i\}). \tag{149}$$

    * *soft attention* corresponds to interpreting $\{\alpha_{ti}\}$ as probabilities of locations
    * *hard attention* corresponds to picking a single location, or non-zero α_{ti}
      · hard attention gradient descent not effective
      · one proposal is to use reinforcement learning

- Spatial transformer module [18]:
  - *the spatial transformer module*, included in a standard neural network architecture transforms deformed inputs to non-deformed ones by providing spatial transformation capabilities
  - the action of the spatial transformer is conditioned on individual data samples, with the appropriate behaviour learnt during training
  - the spatial transformer module is a dynamic mechanism that can actively spatially transform an image by producing an appropriate transformation for each sample
  - the transformation is then performed on the entire feature map (non-locally) and can include scaling, cropping, rotations, as well as non-rigid deformations
  - the spatial transformer has three component:
    * the *localisation network* predicts a transformation to apply to the input image input sample, this is where attention comes is
    * the *grid generator* generates a set of points where the input map should be sampled to produce the transformed output
    * given the feature map and the sampling grid, the *sampler* produces the output map, sampled from the input at the grid points
  - insert spatial transformer into a classification network
- Dense trajectories in action recognition:
  - dense trajectories is a legacy approach to action recognition
  - idea is to find key points & track them using tracklets
    * a tracklet is 15 set of (x, y) coordinates
  - three steps to do this:
    * detect feature points
    * track each key point feature using *optical flow*
      · optical fields provide motion field
    * extract HOG/HOF/MBH time-based features in the coordinate system of each tracklet
      · HOG is generalized to include time
      · HOF is *histogram of flow*
    * the output of histograms was processed by an SVM
- NN & action recognition:
  - one approach to include time, is to extend the convolutional filters, in a CNN, by adding a fourth dimension
    * these filters are referred to as 3D filters to emphasize their spatio-temporal nature,
    * 3D filters are useful for local, short-term events
    * the fusion of time into the network can be done at different levels: slow, early, late or single-frame fusion
  - another approach is to have two networks, one dedicated to images and one to temporal flow, and then fusing the two
  - if global motion need to be captured (i.e. longer than a fraction of a second), then use RNN or LSTM
  - can combine CNN (with or without 3D filters) with LSTMs
  - another proposal is to have each neuron in the CNN to have a local feedback term as well [19]
    * this is similar to using a first order IIR filter rather than a an FIR filer

IX. Natural language processing

- Natural language processing:
  - *natural language processing*, or NLP, is the study of the computational treatment of natural language
    - natural means human
  - the goal of NLP is to be able to design algorithms to allow computers to "understand" natural language to perform certain tasks
- Related fields:
  - linguistics
  - theoretical computer science & AI
  - statistics
  - psychology
- Linguistic Definitions:
  - *syntax* refers to grammatical structure,
  - *semantics* refers to the meaning of the vocabulary symbols arranged with that structure
  - *morphology* is the study of word components, how they are formed, & their relationship to other words in the same language
  - *phonetics* is the study of sound
  - *linguistics* is the scientific study of language and its structure, including the study of morphology, syntax, phonetics, and semantics
  - *lexicon* is the set of words used in a language
  - a *corpus* is a collection of written texts
  - *n-gram* is a contiguous sequence of n items from a given sequence of text or speech
- Phonemes & triphones:
  - a *phoneme* is a single "unit" of sound that has meaning in any language
    - there are 44 phonemes in English
  - a *triphone* is a sequence of three phonemes
    - there are $44^3 = 85,184$ triphones in English
  - because of interference between neighboring phonemes, the 44 phonemes are not sufficiently distinguishable
  - triphones address this interference
- Automatic speech recognition:
  - *automatic speech recognition*, or ASR, addresses the problem of converting spoken speech to text
  - ASR's output is a *transcription* which could be NLP's input
  - conventional ASR is based on GMM-HMM model, described shortly
- Challenges in ASR:
  - natural / conversational speech
  - low SNR
  - speaker variability- age, gender, accents
- Traditional acoustic model:
  - feature extraction, such as spectrogram
  - speaker adaptation
  - phoneme prediction, such as GMM
- Spectrogram:
  - *spectrogram* is a two dimensional, time-frequency representation of voice
  - through signal processing, voice is converted to a spectrogram
  - spectrograms are generated using short term Fourier transforms (SFT)
  - a spectrogram has similarities to an image, but its interpretation is not just a classification problem
  - through filtering, background noise and echoes could be removed from a spectrogram
- GMM:
  - the *Gaussian Mixture Model* (GMM), is a statistical acoustic model

$$\text{spectrogram, or observation vector, } x \xrightarrow{\text{GMM}} \text{feature } p(x|s) \tag{150}$$

    where s is a triphone (class)

  - $p(x|s)$ is assumed to be a mixture of M Gaussian distributions
$$p(x|s) = \sum_{m=1}^{M} c_{sm}\, N(x; \mu_{sm}, C_{sm}) \tag{151}$$
    where for each component m & class s, there is an associated weight c_{sm}, mean μ_{sm} and covariance C_{sm}
- GMM training:
  - the GMM parameters are trained iteratively using the expectation maximisation (EM) algorithm
  - the expectation step estimates $p_m(x|s)$ for all m & the data points x_i in class s
  - the maximization step updates the parameters of the model using the following formulas:
$$\begin{aligned}\mu_{sm} &= \frac{\sum_{i=1}^{N} p_m(x_i|s)\, x_i}{\sum_{i=1}^{N} p_m(x_i|s)} \\ C_{sm} &= \frac{\sum_{i=1}^{N} p_m(x_i|s)\,(x_i-\mu_{sm})(x_i-\mu_{sm})^T}{\sum_{i=1}^{N} p_m(x_i|s)} \\ c_{sm} &= \frac{1}{N}\sum_{i=1}^{N} p_m(x_i|s)\end{aligned} \tag{152}$$
    where N is the number of training points that belong to the class j
- HMM:
  - hidden Markov-model (HMM) was a breakthrough technology in speech recognition during the 1970's
  - an HMM considers triphones as its states s, and reconstruct words & sentences
  - the trellis of HMM is constrained, i.e. not fully connected
  - using merged/shared states, the triphone states are reduced from
$$85,184 \rightarrow 10,000 \tag{153}$$
  - HMMs are the most complex part of an ASR system
  - training the model involves estimating the transition probabilities, and the emission probability density functions
    * this is usually performed by an instance of the EM algorithm known as the Baum-Welch algorithm
    * the expectation part of Baum-Welch algorithm is the BCJR algorithm
- HMM-DNN:
  - starting around 2011, DNNs replaced GMMs as acoustic models
    * DNN replaced the whole acoustic model
  - a DNN generates posterior probabilities $p(s|x)$ rather than $p(x|s)$
  - this hybrid system dominates current ASR
- HMM-free RNN recognition:
  - an active research area is the replacement of HMMs with RNNs
    * RNNs seem to work well as a replacement for both DNN and HMM
  - phonemes can be replaced by characters as the building block
  - the acoustic model outputs $p(a|s)$ where a is a character
  - blanks and junk are represented as underscore
  - using dynamic programming, character sequences are converted to words, multiple characters are collapsed, etc.
- Word vectors:
  - the English vocabulary V is around $|V| = 1$ million words
  - if each word forms a vector basis we end up with a one million dimensional space
  - since words are related, or correlated, the space could be shrunk to $d \ll 10^{13}$ dimensions
    * $d = 100 - 500$ is a good range
    * each dimension would encode some meaning that we transfer using speech
    * this is known as distributed representation of a word
  - thus, we will treat words as a real valued vector
$$w \in \mathbb{R}^d \tag{154}$$
    called *word vectors*
  - correlation between vectors is a measure to how related these words are
- SVD based methods to generate word vectors:

  - see Appendix XV, for SVD review,
  - word w_j is *near* word w_i if its within some window t away from w_i,
  - given a corpus, construct a $(|V| \times |V|)$ *co-occurrence matrix* X, where X_{ij} is the number of times word w_j was found near w_j,
$$X_{ij} = X_{ji}, \tag{155}$$
  - perform *singular value decomposition* (SVD) on X and pick d vectors based on largest d eigenvectors,
  - this is a global approach and is not practical.
- Language models:
  - a *language model* computes the joint probability associated with a sequence of words
$$p(w_1, \cdots, w_n), \tag{156}$$
    * consistent sentences are associated with high probabilities,
  - the *unigram* model is very simple,,
$$p(w_1, \cdots, w_n) = \prod_{i=1}^{n} p(w_i), \tag{157}$$
  - the *bigram* model is a Markov chain with single memory,
$$p(w_1, \cdots, w_n) = \prod_{i=2}^{n} p(w_i|w_{i-1}). \tag{158}$$
  - more generally, the conditional probability is conditioned over a causal or anti-causal window m,
$$p(w_t|w_{t-m}, \cdots, w_{t-1}, w_{t+1}, \cdots, w_{t+m}). \tag{159}$$
- Word vectors to conditional probabilities:
  - given word vectors u and v, the conditional probability $p(u|v)$ is assumed to be of the form,
$$p(u|v) = \frac{\exp(u^T v)}{\sum_{i=1}^{|V|} \exp(u_i^T v)}, \tag{160}$$
  - in other words, the softmax activation (46) of the inner products of two word vectors, is the conditional probability.
- Continuous Bag of Words Model:
  - both the Continuous Bag of Words Model and Skip-Gram model were proposed by *Mikolov* et al.,
  - in *Continuous Bag of Words Model* (CBOW) model, the center word is predicted from the surrounding context,
  - using ML criterion (238) and (159), minimize
$$\min[-\log p(w_t|w_{t-m}, \cdots, w_{t+m})], \tag{161}$$
  - define
$$\hat{v} \triangleq \frac{w_{t-m} + \cdots w_{t-1} + w_{t+1} + \cdots w_{t+m}}{2m}, \tag{162}$$
  - as the name CBOW implies, the order of words in the window does not influence the projection $\hat{v}$,
  - relabeling $u \triangleq w_t$, the ML criterion (161) is approximated to
$$\min[-\log p(u|\hat{v})], \tag{163}$$
  - substituting in (160), we want to minimize,
$$\begin{aligned} &-\frac{\exp(u^T \hat{v})}{\sum_{i=1}^{|V|} \exp(u_i^T \hat{v})} \Rightarrow \\ &-u^T \hat{v} + \log \sum_{i=1}^{|V|} \exp(u_i^T \hat{v}), \end{aligned} \tag{164}$$
  - this can be implemented by a single hidden layer that holds $\hat{v}$ and with an output layer with softmax activation (46).
- Skip-Gram model:
  - the *Skip-Gram model*, the surrounding words are predicted from the center word,

- the objective function J, that we want to minimize becomes

$$\begin{aligned} & -\log p(w_{c-m}, \cdots w_{c+m} | w_c) \\ & \approx -\log \prod_{j=0, j\neq m}^{2m} p(w_{c-m+j}|w_c) \\ & = -\log \prod_{j=0, j\neq m}^{2m} p(u_{c-m+j}|v_c) \\ & = -\log \prod_{j=0, j\neq m}^{2m} \frac{\exp(u_{c-m+j}^T v_c)}{\sum_{i=1}^{|V|} \exp(u_i^T v_c)} \\ & = -\sum_{j=-m, \neq 0}^{m} u_{c+j}^T v_c + 2m \log \sum_{i=1}^{|V|} \exp(u_i^T v_c). \end{aligned} \tag{165}$$

- given the objective function (165), and a window, the gradient w.r.t each center vector v and outside vectors u can be computed,
- collectively denote the parameters that we can optimize as $\sigma \in \mathbb{R}^{2d|V|}$,
  * each word is associated with two vectors, one center and one outside,
- then,

$$\theta^{\text{new}} = \theta^{\text{old}} - \eta \nabla_\theta J(\theta). \tag{166}$$

- Negative sampling:
  - since $|V| \gg 1$ in (165), the resulting algorithm is not practical,
  - instead of the softmax function, can train a binary logistic regression for the center word, and a randomly picked k few word vectors to push down the probability,
  - then a cost function that is simpler than (165), can be used,

$$-\log \theta(u_{c-m+j}^T v_c) - \sum_{j=1}^{k} E_{j\sim p(w)} \log \theta(-u_j^T v_c), \tag{167}$$

  - note that $\theta(-x) = 1 - \theta(x)$,
  - pick $p(w)$ to be the unigram distribution (157) to the power $3/4$ to amplify rare occurrences.
- GloVe model:
  - the GloVe model is an alternate approach of NLP modelling where global and iterative features are combined,
  - the cost function is

$$J(\sigma) = \frac{1}{2} \sum_{i,j=1}^{|V|} f(P_{ij})(u_i^T v_j - \log P_{ij})^2, \tag{168}$$

    where $u_i^T v_j$ is a proxy for P_{ij},
  - can think of the f function as a linear function that saturates for very high frequent words.
- Neural network language model:
  - with neural networks, both the weights and code vectors are being optimized and are parameters,
  - as a result, overfitting is a serious concern,
  - various architectures for *neural network language model* (NNLM),
  - effective models include deep RNNs, and RNTN.
- Recursive networks:
  - recurrent neural networks are based on a chain graph, while recursive networks form a parsing tree,
  - unfortunately both systems are abbreviated by RNN,
    * when ambiguous, will refer to them as recurrent NN and recursive NN
  - a sentence is better modeled using a parsed tree rather than a chain,
    * recursion is helpful in describing natural language,
  - since chains are special case of a tree, recursive NN's are a superset of recurrent NN's.
- Principle of compositionality:
  - phrases, like words, can be mapped to vectors,
  - phrase vectors are generated using word vectors in that phrase, and combining them using some rule.

- Standard recursive neural network:
  - given a parsed tree, a recursive neural network progresses from the leaves to the root of the tree,
    - recursive NN's require a parser to get tree structure,
    - when forming a parsing tree, the greedy algorithm can be used, using a score metric,
    - the *greedy algorithm* recursively makes locally optimal choices at each stage with the hope of finding a global optimum,
  - if p_1 and p_2 are two vectors associated with two phrases, then the concatenated phrase, represented by vector q, is modeled by a single layer neural network,

$$\begin{aligned} q &= \tanh(W_1 p_1 + W_2 p_2 + b), \\ s &= U^T p, \end{aligned} \tag{169}$$

    where $W_1, W_2 \in \mathbb{R}^{d\times d}$, $p_1, p_2, b, q \in \mathbb{R}^{d\times 1}$,
  - the associated score is s,
  - recursive implies all nodes of the tree would use the same W_1, W_2,
  - the rule (169) can be expressed in a more compact form

$$q = \tanh(Wp + b), \tag{170}$$

    where

$$W \triangleq (W_1, W_1), \qquad p \triangleq \begin{pmatrix} p_1 \\ p_2 \end{pmatrix}. \tag{171}$$

- Sentiment:
  - one application of recursive neural network is to classify a word, phrase, or sentence to a sentiment or a rating,
  - the n-way sentiment y^a associated with sentence a, can be extracted using the softmax function (46),

$$y^a = \text{softmax}(W_s a), \tag{172}$$

    where $W_s \in \mathbb{R}^{n\times d}$, and d is the dimensionality of a.
- Determining tree structures using score:
  - the score $s(x, y)$ associated with tree y, and sentence x, is the sum of the parsing decision scores at each node (169),

$$s(x, y) = \sum_{n \in \text{nodes}(y)} s_n, \tag{173}$$

  - the training set is $\{x_i, y_i\}$.
- Recursive RNN variations:
  - Syntactically-Untied, or SU-RNN uses different weights W depending on verb, noun etc.,
  - Matrix-Vector, or MV-RNN associates a word with both a vector and a matrix,
    - the matrix is used to pre-multiply the vector of another phrase vector.
- Recurrent neural tensor networks:
  - *recurrent neural tensor networks*, RNTN were used for sentimental classification,
  - the goal is to generalize (170) by incorporating multiplicative interactions in the update rule to allow a greater interactions between the input vectors,
  - given tensor $V \in \mathbb{R}^{2d\times 2d\times d}$, the generalized update rule (170, 171) is

$$q = \tanh(p^T V p + Wp + b), \tag{174}$$

  - when $V = 0$ the standard recursive neural network is obtained.
- CNN for NLP:
  - the idea is to compute all phrase combinations rather than using a parser,
  - the convolution operator of a CNN is well suited for considering multiple parsing structures in parallel,
    - a convolutional layer computes all sequential h word-vectors,
    - a single convolution operation involves a filter $w \in \mathbb{R}^{hd}$, which is applied to a window of h words to produce a scalar feature c_i,
    - if CNN can handle n words then a feature map $\boldsymbol{c} \in \mathbb{R}^{n-h+1}$,
    - multiple variations are possible,
  - a max pooling layer picks the most important feature by the function,

$$\hat{c} = \max\{\boldsymbol{c}\}, \tag{175}$$

  - will zero-pad shorter phrases,
  - a CNN would use multiple (100s of) different filters, with varying window sizes, resulting in multiple features,
  - the pooled layer outputs are passed to a fully connected softmax layer whose output is the probability distribution over labels.
- Machine translation:
  - consider
    * English language e with model $p(e)$,
    * French language f, with translation model $p(f|e)$,
  - using Bayes' rule, French to English translation can be interpreted as maximizing

$$\max_e p(e|f) = \max_e p(f|e)p(e), \tag{176}$$

  - i.e., given f, using $p(f|e)$ generate multiple candidates for e's, then a decoder computes the most probable joint probability.

X. Tools

- GPU:
  - GPU is an effective hardware that accelerates parallel computations,
  - a GPU is composed of multiple *streaming multiprocessors* or SMs,
    * each SM includes multiple simple processors and a memory,
  - GPUs need to be explicitly parallel-programmed,
  - compute capability is usually measured with *floating point operations per second*, or FLOPS.
- CPU:
  - Intel Advanced Vector Extensions (AVX) is a set of instructions for doing Single Instruction Multiple Data (SIMD) operations on a CPU,
    * AVX2 allows for 32 single precision FLOPS per second, per core
    * power consumption is a concern and AVX2 is usually operated at lower frequencies,
    * at 2500 MHz, AVX2 generates 80 GFLOPS,
  - FMA instruction set is an extension to SIMD instruction set to perform *fused multiply add* (FMA) operations,
  - AVX2 together with FMA could generate up to 500 GFLOPS of computation from a single CPU.
- GPU versus CPU:
  - GPUs optimize throughput whereas CPU optimizes latency,
  - GPUs use more, but simpler processors than CPUs,
  - FLOPS:
    * CPUs ~ 500 GFLOPS,
    * GPUs ~ 10 TFLOPS.
- GPU interface:
  - fastest GPU interface is PCIe3 $\times 16$,
    * with PCIe3 each channel toggles at 8 GT/s,
    * over 16 channels thats 16 GB/s,
    * compared to 10 TFLOPs 16 GB/s may not be sufficient,
  - can connect multiple GPUs using high-performance computer-networking communications standard such as *Infini-Band*(IB).
- NVIDIA Titan X:
  - GPU architecture is Pascal,
  - 3584 NVIDIA CUDA cores,
  - Graphics Card Power is 250 W,
  - GPU engine base clock is 1417 MHz,
  - memory 12 GB GDDR5X,
  - 11 TFLOPs.
- System software i.e. drivers for GPU,
  - CUDA:
    * NVIDIA's CUDA is general purpose parallel computing tool that allows the user to write a single C code that runs on both CPU and GPU,

    * https://developer.nvidia.com/deep-learning/getting-started,
    * higher-level APIs: cuBLAS, cuFFT, cuDNN, etc, see below.
  - OpenCL:
    * similar to CUDA, but runs on any GPU brand,
    * usually slower than CUDA.
- CPU $\leftrightarrow$ GPU interactions in CUDA:
  - CPU is in charge,
  - each CPU and GPU have their dedicated memory,
  - CPU moves data back and forth the two memories using the command *cudaMemcpy*,
  - CPU allocates memory on GPU memory using command *cudaMalloc*,
  - GPU memory capacity may be limited,
  - the host launches kernels on the device.
- Parallel computing using CUDA
  - a *thread* is one path of execution through the code,
  - a CUDA command is of the form
    * name<<<blocks, threads>>>(d-out,d-in)
  - number of threads per block is 512 on older GPUs, and 1024 on newer GPUs,
  - GPU is responsible of allocating blocks to SMs,
  - the blocks and threads are in general three-dimensional entities denoted by

$$\text{dim3}(x, y, z), \tag{177}$$

  - a *map* operation with arguments elements and function, applies the function element-wise on each element.
- Data versus model parallelism:
  - with *data parallelism*, different data $\boldsymbol{x}_i$, are sent to different GPUs,
  - with *model parallelism*, the ANN is partitioned over multiple GPUs.
- With distributed asynchronous gradient descent, different workers don't wait on other events to complete but continue processing data asynchronously.
- BLAS:
  - the BLAS (Basic Linear Algebra Subprograms) are low-level routines that provide standard building blocks for performing basic vector and matrix operations,
  - they are the de facto standard low-level routines for linear algebra libraries,
  - BLAS implementations take advantage of special floating point hardware,
  - many numerical software applications use BLAS-compatible libraries to do linear algebra computations, including cuBLAS, Mathematica, MATLAB, NumPy, and R.
- Software package trends:
  - many of the programming frameworks have performance libraries that harness the HW acceleration,
  - NVIDIA's cuDNN, cuBLAS are optimized for DNN,
    * cuDNN provides a common set of tools that higher level frameworks can use/reuse,
    * cuDNN identifies the 5% of code that (such as convolutions, pooling, activation) takes 80% of run time and delegates the execution of that code to GPU,
- List of software packages:
  - the four major packages are Caffe, Torch, Theano, Tensorflow,
  - Kaldi, is specialized to speech recognition toolkit, & is written in C++,
  - NVIDIA's DIGITS, is an interactive system that provides a quick design capability and visual monitoring tools.
- Caffe:
  - http://caffe.berkeleyvision.org/,
  - developed by U.C. Berkeley,
  - written in C++ / CUDA,
    * do not need to write code to train,
    * Python and Matlab interfaces optional,
    * need to write C++ / CUDA for new GPU layers,
  - popular for CNN users,
  - four main classes:
    * *blobs* are tensors that store data, weights & activations,

      · both data and gradients (diffs) are stored,
    * *layers* interact with bottom blobs & top blobs,
    * a *net* is a bunch of layers, or a graph,
    * a *solver* runs the net forward and backward,
  - makes use of *protocol buffers*, (.proto) to define, for example, net & solver
  - many pre-trained models available through *model zoo*,
    * good for fine-tuning existing networks.
  - not great for RNNs,
  - cumbersome for big networks (GoogLeNet, ResNet).
- Torch:
  - developed at NYU,
  - written in C & Lua,
    * Python and Matlab interfaces optional,
  - used & maintained by Facebook & Twitter,,
  - uses tensors that are very similar to NumPy,
    * in addition to tensor, the nn module lets you easily build and train neural nets,
  - unlike NumPy, GPU is just a data-type away,
    * very easy to run code that runs on GPU,
  - uses *modules* instead of net/layers (Caffe),
  - many pre-trained models available,
  - not great for RNNs.
- Theano:
  - http:/deeplearning.net/software/theano/,
  - developed at University of Montreal,
  - *Theano* is a Python library for symbolic math, with compiler
    * is built on top on NumPy, but also shares similarities with SymPy
    * its some combination of a programming language, a compiler and Python library,
    * embraces computational graphs,
    * automatically computes gradients, through symbolic differentiation,
  - Theano can run on either a CPU or a GPU/CPU,
    * C/C++ compiler on CPU
    * CUDA/OpenCL compiler for GPU,
    * using g++ compiler from TDM-GCC-64 for CPU, see http://tdm-gcc.tdragon.net.
  - a *shared variable* lives in computational graph & persist call to call,
    * shared variables do not need to be identified as inputs in a function,
  - *Keras* and *Lasagne* are higher level wrappers around Theano, or Tensorflow
    * raw Theano is somewhat low-level,
    * Lasagne sets up weights & writes update rules for you,
    * Keras is more high level than Lasagne,
  - works well for RNN,
  - large models can have long compile times,
  - fatter than Torch; more magic,
  - patchy support for pre-trained models.
- Tensorflow is discussed separately.

XI. VC Generalization Bound

- Hoeffding's inequality:
  - *Hoeffding*'s inequality addresses the conditions under which generalization (68) holds,
  - Hoeffding's inequality is a form of large number theory that states

$$P[|C_{\text{in}}(h) - C_{\text{out}}(h)| > \epsilon] \leq 2e^{-2\epsilon^2 N} \tag{178}$$

  - the right hand side of (178) does not depend on $C_{\text{out}}(h)$,
  - Hoeffding's inequality can be applied during testing but not during learning.

- Bounds with M hypotheses:
  - when learning, all M hypotheses may be considered,
  - we do not want any of the M hypothesis deviate from $E(h)_{\text{out}}$,
  - then whatever hypothesis g we chose we are OK,
  - using the union bound, the probability that final hypothesis g is bad (i.e. not tracking) is upper bounded as

$$P[|C_{\text{in}}(g) - C_{\text{out}}(g)| > \epsilon] \leq 2Me^{-2\epsilon^2 N}, \tag{179}$$

  - it follows that as $M \to \infty$ the inequality gets looser.
- Dichotomies & growth functions:
  - N-tuple $(h(\boldsymbol{x}_1), \cdots h(\boldsymbol{x}_N))$ for classifier $h \in \mathcal{H}$, applied on a finite samples $\boldsymbol{x}_1, \cdots \boldsymbol{x}_N \in \mathcal{X}$, is called a *dichotomy*,
    - in other words, a dichotomy is an allowed hypothesis that assigns each vector to a one or a zero,
  - the *growth function* $m_{\mathcal{H}}(N)$, is the maximum number of distinct dichotomies that can be generated with wisely chosen set of N vectors,

$$m_{\mathcal{H}}(N) \triangleq \max_{\boldsymbol{x}_1, \cdots \boldsymbol{x}_N \in \mathcal{X}} |\mathcal{H}|, \tag{180}$$

  - the growth function is upper bounded by

$$m_{\mathcal{H}}(N) \leq 2^N, \tag{181}$$

  - e.g., for the set of convex function hypotheses,

$$m_{\mathcal{H}}(N) = 2^N. \tag{182}$$

- Break points $\Rightarrow$ polynomial growth functions:
  - the smallest value of N for which $m_{\mathcal{H}}(N) \neq 2^N$ is called the *break point*,
  - theorem: if the break point is less than infinity then $m_{\mathcal{H}}(N)$ is polynomial in N,
  - more specifically, *Sauer's Lemma* states that given a break point k,

$$\boxed{m_{\mathcal{H}}(N) \leq \sum_{i=0}^{k-1} \binom{N}{i}} \tag{183}$$

  - note that there are no middle points; $m_{\mathcal{H}}(N) = 2^N$, or $m_{\mathcal{H}}(N)$ is polynomial in N of order $(k-1)$.
- Vapnik-Chervonenkis (VC) Inequality:
  - the *Vapnik-Chervonenkis Inequality*,

$$P[|C_{\text{in}}(g) - C_{\text{out}}(g)| > \epsilon] \leq 4m_{\mathcal{H}}(2N)e^{-\frac{1}{8}\epsilon^2 N} \tag{184}$$

    is a tighter bound than (179),
  - compared to (179), the growth function $m_{\mathcal{H}}(N)$ is substituted for M,
  - VC inequality implies that as long as there is a break point, generalization from in sample to out of sample is possible.
- VC dimension:
  - the *VC dimension* of a hypothesis set $\mathcal{H}$, denoted by $d_{VC}(\mathcal{H})$, is equivalent to break point, where

$$d_{VC}(\mathcal{H}) \triangleq k - 1, \tag{185}$$

  - $d_{VC}(\mathcal{H})$ is finite $\Rightarrow g \in \mathcal{H}$ will generalize,
    - independent of the learning algorithm,
    - independent of the input distribution,
    - independent of target function,
  - VC dimension is the effective binary degrees of freedom of a hypothesis,
    - it measures the effective number of parameters,
  - observation: larger VC dimensions need larger example set N to learn,
  - rule of thumb is to use $N \approx 10\, d_{\text{VC}}$.
- VC generalization bound:

- denote the right hand side of (184) to be δ,

$$\begin{aligned} \delta &\triangleq 4m_{\mathcal{H}}(2N)e^{-\frac{1}{8}\epsilon^2 N} \Rightarrow \\ \ln \frac{\delta}{4m_{\mathcal{H}}(2N)} &= -\tfrac{1}{8}\epsilon^2 N \Rightarrow \\ \epsilon &= \sqrt{\frac{8}{N}\ln\frac{4m_{\mathcal{H}}(2N)}{\delta}} \\ &\triangleq \Omega, \end{aligned} \tag{186}$$

  where Ω is called the *generalization error*,
- it follows that with probability $\geq 1-\delta$,

$$\begin{aligned} |C_{\text{out}} - C_{\text{in}}| &\leq \Omega(N, \mathcal{H}, \delta) \Rightarrow \\ C_{\text{out}} - C_{\text{in}} &\leq \Omega \Rightarrow \\ C_{\text{out}} &\leq C_{\text{in}} + \Omega, \end{aligned} \tag{187}$$

- theorem: for any tolerance $\delta > 0$, with probability $\geq 1-\delta$,

$$\boxed{C_{\text{out}}(g) \leq C_{\text{in}}(g) + \sqrt{\frac{8}{N}\ln\frac{4m_{\mathcal{H}}(2N)}{\delta}}} \tag{188}$$

- with a larger hypothesis set, we get a better approximation (smaller C_{in}) but a worst generalization error (larger Ω).

• Classification versus estimation:
  - VC generalization bound can be generalized from classification problems to also include estimation problems,
  - bias and variance analysis is an alternative to VC analysis, that is suited when squared error measure is used as an error metric.

XII. Bias & Variance Method

• Average hypothesis:
  - let $g^{(\mathcal{D})}(\boldsymbol{x})$ be the final hypothesis associated with data set $\mathcal{D}$,
  - the *average hypothesis* $\bar{g}(\boldsymbol{x})$ is defined as

$$\bar{g}(\boldsymbol{x}) \triangleq E_{\mathcal{D}}[g^{(\mathcal{D})}(\boldsymbol{x})], \tag{189}$$

  - in general $\bar{g}(\boldsymbol{x})$ is not known,
  - $\bar{g}(\boldsymbol{x})$ is a function of the specific learning algorithm,
  - given N, it is assumed that $\bar{g}(\boldsymbol{x})$ is the best hypothesis a family of hypothesis $\mathcal{H}$ can generate.

• Bias or deterministic noise:
  - define *bias* as

$$\text{bias} \triangleq E_x[(\bar{g}(\boldsymbol{x}) - f(\boldsymbol{x}))^2], \tag{190}$$

  - the bias term is the part of the target function f that $\mathcal{H}$ can not capture,
  - an alternative name is *deterministic noise*,
  - bias is related to approximation, i.e. on how well the hypothesis fits the data.

• Bias & variance method:
  - *bias and variance* analysis is an alternate decomposition of C_{out},

$$C_{\text{out}}(g^{\mathcal{D}}) \triangleq E_x[(g(\boldsymbol{x}) - f(\boldsymbol{x}) - n)^2] \tag{191}$$

  - instead of C_{in}, the decomposition is w.r.t. the average hypothesis $\bar{g}(\boldsymbol{x})$,
    * as a result, the bias-variance analysis depends on the learning algorithm,
  - the *variance* term,

$$\text{var} \triangleq E_x[E_{\mathcal{D}}(g^{(\mathcal{D})}(\boldsymbol{x}) - \bar{g}(\boldsymbol{x}))^2], \tag{192}$$

  is the variation in error due to different data sets and the corresponding hypotheses choices,
  - let σ_n^2 denote the variance of stochastic noise,
  - by introducing $\bar{g}(\boldsymbol{x})$ inside (191), by taking expectations w.r.t. all data sets $\mathcal{D}$, and by substituting (189, 190, 192) into (191), we obtain

$$E_{\mathcal{D}}[C_{\text{out}}(g^{\mathcal{D}})] = \text{var} + \text{bias} + \sigma_n^2, \tag{193}$$

- the bias & variance analysis decomposes C_{out} to three components:
  1) the variance term quantifies the distance from a hypothesis $g^{(\mathcal{D})}$ to optimal hypothesis $\bar{g}$,
  2) the bias term quantifies the distance from optimal hypothesis $\bar{g}$ to target function f,
  3) the noise term quantifies the distance target function f to observed quantities y,
- bias-variance analysis is a conceptual tool; in practice the terms can not be computed.

XIII. SVM'S & KERNEL METHODS

- Support vector machine classifiers:
  - a *support vector machine*, or SVM, is a learning algorithm for the perceptron
  - SVM is a classifier based on supervised training
  - two other classifiers with supervised learning are covered in Section XV:
    * naive Bayes
    * decision tree
- Hard versus soft margin:
  - *hard margin* SVM implies training data is linearly separable
  - *soft margin* SVM implies training data is not linearly separable
  - will start with *hard margin*, where in general, an infinite number of linear solutions classify data correctly
- Distance:
  - relabel & shorten $\boldsymbol{w}$ by pulling out the bias $b = w_0$, and dropping $x_0 = 1$
    * the resulting vector $\boldsymbol{w}$, which represents the decision hyperplane, is orthogonal to the hyperplane itself
  - define the *distance* $d(\boldsymbol{x}_n, \boldsymbol{x})$, between any $\boldsymbol{x}_{\text{n}} \in \mathbb{R}^d$ & an arbitrary $\boldsymbol{x}$ on hyperplane, as the projection of $(\boldsymbol{x}_{\text{n}} - \boldsymbol{x})$ orthogonal to the decision hyperplane

$$\begin{aligned} d(\boldsymbol{x}_n, \boldsymbol{x}) &\triangleq \left| \frac{\boldsymbol{w}^T}{\|\boldsymbol{w}\|} \cdot (\boldsymbol{x}_{\text{n}} - \boldsymbol{x}) \right| \\ &= \frac{1}{\|\boldsymbol{w}\|} \left| \boldsymbol{w}^T \boldsymbol{x}_{\text{n}} + b - \boldsymbol{w}^T \boldsymbol{x} - b \right| \\ &= \frac{|\boldsymbol{w}^T \boldsymbol{x}_{\text{n}} + b|}{\|\boldsymbol{w}\|} \end{aligned} \tag{194}$$

- Margin:
  - an SVM determines the weight vector $\boldsymbol{w}$ in order to maximize the distance from the decision hyperplane to the nearest point(s)
  - the minimum distance is called the *margin*
  - with SVM, the *margin* around a hyperplane is maximized
  - similar to minmax
- Support vectors:
  - the nearest vectors (points) that define the margin are called the *support vectors* (SV)
    * only SVs influence the weights, i.e. learning
    * all other points are ignored
- SVM & complexity d_{VC}:
  - imposing margin around hyperplane reduces the growth function
  - as a result, d_{VC} gets smaller which should help with generalization
- Normalization:
  - normalize $(\boldsymbol{w}, b)$ such that for any $\boldsymbol{x}_{\text{SV}}$

$$|\boldsymbol{w}^T \boldsymbol{x}_{\text{SV}} + b| = 1 \tag{195}$$

  - since all $\boldsymbol{x}_{\text{SV}}$ are at same distance from hyperplane, the above normalization applies to all support vectors
  - when $\boldsymbol{x}_n = \boldsymbol{x}_{SV}$, substituting (195) into (194)

$$\boxed{d(\boldsymbol{x}_{SV}, \boldsymbol{x}) = \frac{1}{\|\boldsymbol{w}\|}} \tag{196}$$

  - the constraint (195) can be re-interpreted as

$$\begin{gathered} \min_{\boldsymbol{x}_n \in \boldsymbol{x}_{\text{SV}}} |\boldsymbol{w}^T \boldsymbol{x}_{\text{n}} + b| = 1 \Rightarrow \\ \forall \boldsymbol{x}_n,\ y_n(\boldsymbol{w}^T \boldsymbol{x}_{\text{n}} + b) \geq 1 \end{gathered} \tag{197}$$

- Margin violation with soft-margin:
  - with soft-margin, data points $(\boldsymbol{x}_n, y_n)$ are permitted to violate the margin by $\xi_n > 0$
  - in other words, from (197), training data $(\boldsymbol{x}_n, y_n)$ violates the margin by $\xi_n > 0$, if

$$y_n(\boldsymbol{w}^T\boldsymbol{x}_{\text{n}} + b) \geq 1 - \xi_n \tag{198}$$

  - the *total violation* is

$$\sum_i \xi_i \tag{199}$$

- SVM optimization:
  - with hard margin SVM, & under constraint (195), maximizing $1/\|\boldsymbol{w}\|$ (196), is equivalent to minimizing

$$C = \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} \tag{200}$$

  - with soft margin SVM, the total violation (199) is also included in the cost function

$$C = \min \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + C\sum_i \xi_i \tag{201}$$

    under the two sets of constraints, $\forall n$

$$\begin{gathered} y_n(\boldsymbol{w}^T\boldsymbol{x}_{\text{n}} + b) \geq 1 - \xi_n \\ \xi_n \geq 0 \end{gathered} \tag{202}$$

  - C is usually determined through cross validation
    - C trades error penalty for stability
    - larger C may minimize error rate on training data but may also overfit
  - the associated Lagrangian $\mathcal{L}(\boldsymbol{w}, b, \boldsymbol{\alpha}, \boldsymbol{\xi})$, with the inequality constraints is known as KKT, and is given by

$$\mathcal{L} = \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + C\sum_n \xi_n - \sum_n \beta_n\xi_n - \sum_n \alpha_n[y_n(\boldsymbol{w}^T\boldsymbol{x}_{\text{n}} + b) - 1 + \xi_n] \tag{203}$$

  - we want to minimize $\mathcal{L}(\boldsymbol{w}, b, \boldsymbol{\alpha}, \boldsymbol{\xi})$ w.r.t $(\boldsymbol{w}, b)$ and $\boldsymbol{\xi}$, but maximize it w.r.t. $\alpha_n \geq 0$
- $\mathcal{L}(\boldsymbol{w}, b, \boldsymbol{\alpha}, \boldsymbol{\xi}) \to \mathcal{L}(\boldsymbol{\alpha})$:
  - differentiating w.r.t $(\boldsymbol{w}, b)$ and $\boldsymbol{\xi}$

$$\begin{gathered} \nabla_\omega \mathcal{L} = \boldsymbol{w} - \sum_n \alpha_n y_n \boldsymbol{x}_n = 0 \\ \frac{\partial \mathcal{L}}{\partial b} = \sum_n \alpha_n y_n = 0 \\ \frac{\partial \mathcal{L}}{\partial \xi_n} = C - \alpha_n - \beta_n = 0 \end{gathered} \tag{204}$$

  - substituting (204) into (203), SVM optimization is reduced to a maximization over $\boldsymbol{\alpha}$

$$\mathcal{L}(\boldsymbol{\alpha}) = \sum_{n=1}^N \alpha_n - \frac{1}{2}\sum_{n,m=1}^N y_n y_m \alpha_n \alpha_m \boldsymbol{x}_n^T \boldsymbol{x}_m^T \tag{205}$$

    under the constraints

$$\begin{gathered} \forall n,\ 0 \leq \alpha_n \leq C \\ \sum_{n=1}^N \alpha_n y_n = 0 \end{gathered} \tag{206}$$

- Quadratic programming, $\mathcal{L}(\boldsymbol{\alpha}) \to \boldsymbol{\alpha}$:
  - the optimization in (205) can be expressed as

$$\min_\alpha \frac{1}{2}\boldsymbol{\alpha}^T Q \boldsymbol{\alpha} - \mathbf{1}^T\boldsymbol{\alpha} \tag{207}$$

    where

$$Q_{ij} \triangleq y_i y_j \boldsymbol{x}_i^T \boldsymbol{x}_j \tag{208}$$

- from (206), the constraints in vector form are

$$\begin{aligned} \mathbf{0} \le \boldsymbol{\alpha} \le \boldsymbol{C} \\ \boldsymbol{y}^T \boldsymbol{\alpha} = 0 \end{aligned} \tag{209}$$

- there are tools available to solve such optimization problems, known as *quadratic programming*
- since Q is $(N \times N)$, such an optimization can become problematic for large N

- Classes of $\boldsymbol{\alpha}$:
  - $\alpha_n = 0 \Rightarrow \boldsymbol{x}_n$ is an interior point,
  - $0 < \alpha_n < C \Rightarrow \boldsymbol{x}_n$ is a *margin support vector*, with $\xi_n = 0$
  - $\alpha_n = C \Rightarrow \boldsymbol{x}_n$ is *non-margin support vector*, with $\xi_n > 0$
- $\boldsymbol{\alpha} \to \boldsymbol{w}, b$:
  - given $\boldsymbol{\alpha}$, $\boldsymbol{w}$ is determined from (204)
  - the KKT condition implies $\alpha_n = 0$ for all n that are not support vectors,
  - as a result $\boldsymbol{w}$ can be expressed as

$$\boldsymbol{w} = \sum_{\alpha_n > 0} \alpha_n y_n \boldsymbol{x}_n \tag{210}$$

  - in PLA (48), all points that mismatch are updated whereas in (210) only the SV are used to derive $\boldsymbol{w}$
  - the b parameter can be solved from any SV, see (203)

$$\begin{aligned} y_i(\boldsymbol{w}^T \boldsymbol{x}_i + b) = 1 \Rightarrow& \\ b &= \frac{1}{y_i} - \boldsymbol{w}^T \boldsymbol{x}_i \\ &= \frac{1}{y_i} - \sum_{\alpha_n > 0} \alpha_n y_n \boldsymbol{x}_n^T \boldsymbol{x}_i \end{aligned} \tag{211}$$

- SVM in feature space & the kernel:
  - consider a nonlinear space mapping

$$x \in \mathbb{R}^n \to z \in \mathbb{R}^m \tag{212}$$

    * in general $m > n$
    * the goal for this mapping is to make a classification problem linearly separable in the z-space, whereas it was not in the x-space
  - optimization in z-space proceeds similar to (207, 209), but with the substitution $\boldsymbol{x}_i \to \boldsymbol{z}_i$
  - since sample size N does not change, the only difference is the change in the inner products

$$\boldsymbol{x}_i^T \boldsymbol{x}_j \to \boldsymbol{z}_i^T \boldsymbol{z}_j \tag{213}$$

  inside the Lagrangian (208)
    * only the inner product of $\boldsymbol{z}$ is needed to do the computations, not $\boldsymbol{z}$ itself
    * the dimensionality of $\boldsymbol{z}$ is not important
    * this inner product $\boldsymbol{z}_i^T \boldsymbol{z}_j$ is some function of $(\boldsymbol{x}_i, \boldsymbol{x}_j)$, called the *kernel*

$$\boxed{\boldsymbol{z}_i^T \boldsymbol{z}_j \triangleq K(\boldsymbol{x}_i, \boldsymbol{x}_j)} \tag{214}$$

  - when used with non-linear transformations, SVM's generate sophisticated boundaries (or hypothesis h) that are simply generated (i.e. simple $\mathcal{H}$)
- Bound on C_{out}:
  - theorem: given the sample size N and number of SV's, the out-of-sample performance can be bounded

$$E[C_{\text{out}}] \le \frac{E[\#SV's]}{N-1} \tag{215}$$

  where $\#SV's$ is an in-sample determined value that is applied on out-of-sample estimate
  - key takeaway is that this bound is independent of dimensionality d
    * we can transform space so that $\tilde{d} \to \infty$, without impacting C_{out}
  - this is the main theoretical result in support of SVM
- $\boldsymbol{z}_i^T \boldsymbol{z}_j$ is sufficient statistics:
  - it was observed in (213) that individual $\boldsymbol{z}_i$ vectors need not to be known to optimize the Lagrangian, just their inner product

  - i.e., when using quadratic programming the Q matrix components (208), become

$$Q_{ij} \triangleq y_i y_j K(\boldsymbol{x}_i, \boldsymbol{x}_j) \tag{216}$$

  - similarly, only $\boldsymbol{z}_i^T \boldsymbol{z}_j$ is needed to compute g, since from (210, 211), $g(\boldsymbol{x})$ is given by

$$\begin{aligned} &\text{sign}(\boldsymbol{w}^T \boldsymbol{z} + b) \\ =&\text{sign}\left(\sum_{\boldsymbol{z}_n \text{is SV}} \alpha_n y_n (\boldsymbol{z}_n^T \boldsymbol{z} - \boldsymbol{z}_n^T \boldsymbol{z}_i) + \frac{1}{y_i} \right) \\ =&\text{sign}\left(\sum_{\alpha_n > 0} \alpha_n y_n (K(\boldsymbol{x}_n, \boldsymbol{x}) - K(\boldsymbol{x}_n, \boldsymbol{x}_i)) + \frac{1}{y_i} \right) \end{aligned} \tag{217}$$

  - note that the signal goes through two nonlinearities, K and the sign()
  - as long as we have access to $K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \boldsymbol{z}_i^T \boldsymbol{z}_j$, we do not need to know about z-space
- Kernel requirements:
  - two characteristics desired from a kernel:
    * the kernel should be valid, i.e., represent the inner product in some $\mathcal{Z}$ space
    * the kernel should be a good match for the learning problem in hand
      · hand picking a kernel is a disadvantage compared to deep learning
  - theorem: Mercer's condition, a kernel is valid iff for any $\boldsymbol{x}_1, \cdots, \boldsymbol{x}_N$
    * $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$ is symmetric, and
    * the $(N \times N)$ matrix with coefficients $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$ is positive semi-definite
- Kernel examples:
  - kernels of space $\mathcal{Z}$, that supports polynomial order Q, can be computed efficiently in the form

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = (b + a\boldsymbol{x}_i^T \boldsymbol{x}_j)^Q \tag{218}$$

    where a and b are scale factors
  - the *radial basis function* (RBS) kernel

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j) \triangleq e^{-\gamma \|\boldsymbol{x}_i - \boldsymbol{x}_j\|^2} \tag{219}$$

    is associated with an infinite dimensional $\mathcal{Z}$ space

XIV. Similarity Methods

- Nearest neighbor method:
  - decode any $\boldsymbol{x}$ to its nearest data set of N elements $(\boldsymbol{x}_i, y_i)$
  - in other words, the data output is the same output as its *nearest neighbor* (NN)
  - like ML decoding
  - complexity is proportional to N
- k-nearest neighbor method:
  - find the best k closest neighbors, and make a decision based on the k candidates y_i
  - higher values of k have a smoothing effect that makes the classifier more resistant to outliers
- Bounded distance method:
  - this is similar to bounded distance decoding
  - can think of a cylinder around each training data point
- Radial basis function method:
  - after observing N data samples, the *radius basis function* hypothesis, or RBF, is

$$h(\boldsymbol{x}) = \sum_{n=1}^{N} w_n e^{-\gamma \|\boldsymbol{x} - \boldsymbol{x}_n\|^2} \tag{220}$$

  - the exponents form the basis functions
  - since we have N equations and N unknowns, the exact interpolation solution ends up being

$$\boldsymbol{w} = \Phi^{-1} \boldsymbol{x} \tag{221}$$

    where $\Phi_{ij} = e^{-\gamma \|\boldsymbol{x}_i - \boldsymbol{x}_j\|^2}$
- Bounded distance as an RBF:

- when the basis function is chosen to be the cylinder function rather than Gaussian, RBF becomes bounded distance method
- can think of RBF to be the softened version of bounded distance method

- Similarity methods:
  - the nearest neighbor, bounded distance and the RBF are examples of *similarity methods* since we are comparing how similar data is to the training set
- RBF with K centers:
  - use $K \ll N$ centers instead of N, for better generalization,
  - then the model of (220) changes to
$$h(\boldsymbol{x}) = \sum_{n=1}^{K} w_n e^{-\gamma \|\boldsymbol{x} - \boldsymbol{\mu}_n\|^2}, \tag{222}$$
  - the K centers $\boldsymbol{\mu}_n$ are new parameters, each being a d dimensional vector.
- Weights for K-centers:
  - from (222),
$$\Phi \boldsymbol{w} = \boldsymbol{y}, \tag{223}$$
    where $\Phi_{ij} = e^{-\gamma \|\boldsymbol{x}_i - \boldsymbol{\mu}_j\|^2}$,
  - given the centers $\boldsymbol{\mu}_k$, or Φ, there are N equations and K unknowns in (223),
  - such a system, can be solved similar to linear regression,
  - if $\Phi^T \Phi$ is invertible,
$$\boldsymbol{w} = (\Phi^T \Phi)^{-1} \Phi^T \boldsymbol{y}. \tag{224}$$
- Determining γ through EM:
  - γ in (222)can be determined iteratively, using *expectation-maximization* (EM) algorithm,
    * given γ, solve for $\boldsymbol{w}$,
    * given $\boldsymbol{w}$, minimize error w.r.t. γ.
- RBF versus SVM:
  - when the kernel is chosen as in (219), the SVM solution is in the form, see (217)
$$\text{sign}\left(\sum_{\alpha_n > 0} \alpha_n y_n e^{-\gamma \|\boldsymbol{x}_i - \boldsymbol{x}_j\|^2} + b \right), \tag{225}$$
  - similarity with RBF can be noted by comparing (225) to (222).
- Smoothness quantified:
  - smooth function are functions with small derivatives,
  - smoothness can be measured by the metric
$$\sum_{k=0}^{\infty} a_k \int_{-\infty}^{\infty} \left(\frac{d^k h}{dx^k} \right) dx. \tag{226}$$
- RBF and regularization:
  - consider minimizing a function under smoothness (226) constraint,
$$\sum_{n=1}^{N} (h(x_n) - y_n)^2 + \lambda \sum_{k=0}^{\infty} a_k \int_{-\infty}^{\infty} \left(\frac{d^k h}{dx^k} \right) dx \tag{227}$$
  - when this is solved, we get RBF with Gaussian functions.

XV. Miscellaneous Mathematical Entities

- Singular value decomposition:
  - $\forall\ (m \times n)\ \Omega,\ \exists$
    * unitary $(m \times m)\ U$
    * unitary $(n \times n)\ V$ and
    * non-negative real diagonal $(m \times n)\ D$

    such that
$$\Omega = UDV^H \tag{228}$$

  - this factorization is called *singular value decomposition* (SVD)
- SVD singular vectors & singular values:
  - the columns of U, $\boldsymbol{u}_k$, are the eigenvectors of AA^T
    - $\boldsymbol{u}_k$ are called the *left singular vectors*
  - the columns of V, are the eigenvectors of A^TA
    - the columns of V, $\boldsymbol{v}_k$, are called the *right singular vectors*
  - $D_{ii} > 0$ are called the *singular values* of Ω
    - by convention, order diagonal entries such that

$$D_{i,i} \geq D_{i-1,i-1} \tag{229}$$

  - if there are $r \leq \min(m, n)$ singular values, then the rank of Ω in (228) is r
  - D_{ii} are the square roots of the nonzero eigenvalues of both AA^T & A^TA
  - if all singular values of Ω are unique and non-zero, then its SVD is unique
    - this uniqueness is up to the multiplication of a column of U by a unit-phase factor and simultaneous multiplication of the corresponding column of V by the same unit-phase factor
- SVD & vector compression:
  - let $D^{(s)}$ be the matrix derived from D, that keeps only the $s < r$ largest singular values, replacing the rest with zeros
  - from (228), define rank $s < r$ matrix

$$\begin{aligned}\Omega^{(s)} &\triangleq UD^{(s)}V^H \\ &= \sum_{k=0}^{s-1} \boldsymbol{u}_k D_{kk} \boldsymbol{v}_k^T\end{aligned} \tag{230}$$

  - each term $\boldsymbol{u}_k D_{kk} \boldsymbol{v}_k^T$ is called *principle image*
  - $\Omega^{(s)}$ is the closest rank-l matrix to Ω, where the term closest is in term of componentwise Euclidean norm

$$\sum_{i,j} (\Omega_{ij} - \Omega_{ij}^{(s)})^2 \tag{231}$$

  - the property of SVD to provide the closest rank-s approximation for a matrix Ω can be used to reduce vector dimensionality
    - in other words, SVD can be used to compress Ω
  - such a compression is not necessarily the best way to compress images
- Principle component analysis:
  - *principle component analysis*, or PCA, reduces the data set dimensionality from d to $\tilde{d} < d$
  - given $(N \times d)$ data matrix X, first the $(d \times d)$ covariance matrix $\overline{C}$, is computed (10)
  - $\overline{C}$ is symmetric and has a spectral factorization

$$\overline{C} = V\Lambda V^T \tag{232}$$

  - the d dimensional columns of V are the eigenvectors of $\overline{C}$ called *principle axes*
  - the $(d \times d)$ diagonal Λ contains the associated eigenvectors
  - choose $\tilde{d}$ eigenvalues corresponding to the largest $\tilde{d}$ eigenvalues
  - denote the resulting $(\tilde{d} \times d)$ matrix by $\tilde{V}$
  - the projections of the data on the principal axes are called *principal components*
  - any data vector $\boldsymbol{x}$ can be compressed to $\tilde{\boldsymbol{x}}$ by

$$\tilde{\boldsymbol{x}} = \tilde{V}\boldsymbol{x} \tag{233}$$

- Relating SVD to PCA:
  - when $X = \Omega$ (228),

$$\begin{aligned}X^TX &= (UDV^T)^T(UDV^T) \\ &= VDU^TUDV^T \\ &= VD^2V^T\end{aligned} \tag{234}$$

  - comparing (234) to (232),

$$D^2 = \Lambda. \tag{235}$$

- Relative entropy:

- the *relative entropy* of $p(x)$ with respect to the *entropy measure* $q(x)$ is defined as

$$D(p||q) \triangleq \sum_{x\in\mathfrak{X}} p(x)\log\frac{p(x)}{q(x)} = E_p \log\frac{p(X)}{q(X)}, \tag{236}$$

- $q(x)$ does not need to be a probability measure,
- when $q(x) = 1$, the entropy measure is known as the *uniform measure*,
  * with uniform measure, relative entropy reduces to *entropy*,

$$H(X) \triangleq D(X||1), \tag{237}$$

  * in that sense relative entropy generalizes entropy,
- when $q(x)$ is a probability measure, then relative entropy is also called *Kullback Leibler distance* between $p(x)$ and $q(x)$,
  * $D(p||q)$ is a measure of the inefficiency of assuming the distribution is q when when the true distribution is p,
  * $D(p||q) \geq 0$,
  * $D(p||q) = 0 \Leftrightarrow p = q$,
  * $D(p||q)$ is not a true distance since it is not *symmetric* and does not satisfy the *triangle inequality*,
  * $D(p||q)$ is *convex* in the pair (p, q).

- Bayesian criterion:
  - consider the hypotheses set $h \in \mathcal{H}$, & a data set $\mathcal{D}$,
    * practical hypotheses sets are infinite,
  - in the *Bayesian approach*, the posterior $P(h|\mathcal{D})$ is computed from prior $P(h)$, and the maximum $P(h = g|\mathcal{D})$ is chosen,
  - it assumes the prior distribution on hypothesis set $P(h)$, is known,
    * this approach is justified when prior is valid or irrelevant.
- Maximum-likelihood criterion:
  - for independently generated data $\{\boldsymbol{x}_n\}$, the maximum-likelihood (ML) criterion maximizes the expression

$$\begin{aligned}
&\max_{\{\boldsymbol{y}_n\}} \prod_{n=1}^{N} P(\boldsymbol{y}_n|\boldsymbol{x}_n) \Rightarrow \\
&\min_{\{\boldsymbol{y}_n\}} -\log\left(\prod_{n=1}^{N} P(\boldsymbol{y}_n|\boldsymbol{x}_n)\right) \Rightarrow \\
&\min_{\{\boldsymbol{y}_n\}} \frac{1}{N}\sum_{n=1}^{N}\log\frac{1}{P(\boldsymbol{y}_n|\boldsymbol{x}_n)}
\end{aligned} \tag{238}$$

- Naive Bayes model:
  - *Naive Bayes model* is a subclass of Bayesian Network, BN, that makes ”naive assumptions”
  - often used in classification, where given feature observations $X_1, X_2, \cdots, X_n$ per sample, the model *inferences* the class C
  - analytically:

$$\begin{aligned}
&\forall\, i, j \neq i,\ (X_i \perp X_j \mid C) \Rightarrow \\
&P(C, X_1, \cdots, X_n) = P(C)\prod_{i=1}^{n} P(X_i|C) \Rightarrow \\
&P(c_i|x_1, \cdots, x_n) \propto P(c_i)\prod_{j=1}^{n} P(x_j|c_i) \Rightarrow \\
&\hat{c} = \arg\max_{c_i} P(c_i)\prod_{j=1}^{n} P(x_j|c_i)
\end{aligned} \tag{239}$$

  - $P(c_i)$ is determined using the relative frequency of c_i from the training data
  - *Gaussian Naive Bayes* model assumes $P(x_j|c_i)$ is Gaussian
    * use MAP estimation to find (μ_i, σ_i)
- Decision tree:
  - the space is partitioned into sub-regions by sequentially & linearly splitting the space based on single features

- the resulting structure is a decision tree
- a decision is designed to maximize the *information gain*, where

$$\text{information gain} = \text{parent-entropy} - \text{(weighted average) children-entropy} \tag{240}$$

- Cross-entropy:
  - define *cross-entropy* between two probability distributions p & q over the same underlying set of events X to be

$$H(p,q) \triangleq -E_p \log q(X), \tag{241}$$

  - expanding (236),

$$\begin{aligned} D(p||q) &= E_p \log \frac{p(X)}{q(X)} \\ &= E_p \log p(X) - E_p \log q(X) \Rightarrow \\ H(p,q) &= H(p) + D(p||q), \end{aligned} \tag{242}$$

  - the cross entropy between two probability distributions measures the average number of bits needed to identify an event from a set of possibilities.
- ML criterion & cross-entropy:
  - consider the ML criterion (238) with scalar x_n, with $y_n \in \{0,1\}$ and in the limit $N \to \infty$

$$\begin{aligned} &\min_{\{y_n\}} \lim_{N\to\infty} \frac{1}{N} \sum_{n=1}^{N} \log \frac{1}{P(y_n|x_n)} \\ =&\min_{\{y_n\}} \lim_{N\to\infty} \frac{1}{N} \left(\sum_{n:y_n=1} \log \frac{1}{P(y_n=1|x_n)} + \sum_{n:y_n=0} \log \frac{1}{P(y_n=0|x_n)} \right) \end{aligned} \tag{243}$$

  - define two terms

$$\begin{aligned} q(x_n) &\triangleq P(y_n=1|x_n) \\ p &\triangleq \lim_{N\to\infty} \frac{\sum_{n=1}^{N} y_n}{N} \end{aligned} \tag{244}$$

  - substitute $q(x_n)$ into (243)

$$\begin{aligned} &= \min_{p} \lim_{N\to\infty} \left(\frac{1}{N} \sum_{i=1}^{pN} \log \frac{1}{q(x_i)} + \frac{1}{N} \sum_{i=1}^{(1-p)N} \log \frac{1}{1-q(x_i)} \right) \\ &= \min_{p} \left(p\, E_X \log \frac{1}{q(X)} + (1-p) E_X \log \frac{1}{1-q(X)} \right) \end{aligned} \tag{245}$$

  - compare this to minimizing the cross-entropy (241)

$$= \min_{p}\ E_p \frac{1}{\log q(X)} \tag{246}$$

- Linear regression:
  - linear regression has a long history in statistics
  - given dataset $\{(\boldsymbol{x}_n, y_n)\}$, with $y_n \in \mathbb{R}$, find best linear fit that minimizes mean-square error
  - from (25), the in-sample error can be represented as

$$\begin{aligned} C_{\text{in}}(\boldsymbol{w}) &= \frac{1}{N} \|X\boldsymbol{w} - \boldsymbol{y}\|^2 \\ &= \frac{1}{N} (X\boldsymbol{w} - \boldsymbol{y})^T (X\boldsymbol{w} - \boldsymbol{y}) \\ &= \frac{1}{N} (\boldsymbol{w}^T X^T X \boldsymbol{w} - 2\boldsymbol{w}^T X^T \boldsymbol{y} + \boldsymbol{y}^T \boldsymbol{y}) \end{aligned} \tag{247}$$

    where
    - X is the $N \times (d+1)$ data matrix whose rows are the inputs $\boldsymbol{x}_n$
    - $\boldsymbol{y} = \{y_n\}$ is $(N \times 1)$
  - imposing $\nabla C_{\text{in}}(\boldsymbol{w}) = 0$ on (247)

$$\begin{aligned} \nabla(\boldsymbol{w}^T X^T X \boldsymbol{w}) &= \nabla(2\boldsymbol{w}^T X^T \boldsymbol{y}) \Rightarrow \\ X^T X \boldsymbol{w} &= X^T \boldsymbol{y} \end{aligned} \tag{248}$$

  - if $(d+1) \times (d+1)$ $X^T X$ is invertible, a one-shot learning formulation is obtained

$$\begin{aligned} \boldsymbol{w} &= (X^T X)^{-1} X^T \boldsymbol{y} \\ &= X^{\dagger} \boldsymbol{y} \end{aligned} \tag{249}$$

    where

$$X^{\dagger} \triangleq (X^T X)^{-1} X^T \tag{250}$$

    is the pseudo-inverse of X
- Gabor filter:
  - in image processing, a *Gabor filter*, named after Dennis Gabor, is a linear filter used for edge detection,
  - in the spatial domain, a 2D Gabor filter is a Gaussian kernel function modulated by a sinusoidal plane wave.
- K-means clustering:
  - K-means clustering is a method of clustering data into K classes where each cluster is identified by its center $\boldsymbol{\mu}_k$,
  - each center $\boldsymbol{\mu}_k$ is the representative of a group of data, called *cluster* S_k,
  - the goal is to minimize

$$\sum_{k=1}^{K} \sum_{\boldsymbol{x}_n \in S_k} \|\boldsymbol{x}_n - \boldsymbol{\mu}_k\|^2, \tag{251}$$

  - since no y_n is involved in the above minimization, this is an example of unsupervised learning,
    * unsupervised learning does not 'corrupt' the data,
  - in general this is an NP hard problem.
- Lloyd's algorithm:
  - *Lloyd's algorithm* is an iterative solution to the K-means clustering:
    * given S_k, update $\boldsymbol{\mu}_k$,

$$\boldsymbol{u}_k = \frac{1}{|S_k|} \sum_{\boldsymbol{x}_n \in S_k} \boldsymbol{x}_n, \tag{252}$$

    * given $\boldsymbol{\mu}_k$, update S_k so that each sample picks the nearest cluster.

References

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, 2015.
[2] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological Review*, 1958.
[3] C. Szegedy, W. Liu, *et al.*, "Going deeper with convolutions," 2015.
[4] C. Szegedy, S. Ioffe, V. Vanhoucke, *et al.*, "Inception-v4, inception-resnet and the impact of residual connections on learning," 2016.
[5] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradientbased learning applied to document recognition," 1998.
[6] A. Krizhevsky, I. Sutskever, and G. Hinton, "Imagenet classification with deep convolutional neural networks," 2012.
[7] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," 2013.
[8] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2015.
[9] K. He, X. Zhang, S. Ren, and J. Sun, "Identity mappings in deep residual networks," 2016.
[10] S. Hochreiter and J. Schmidhuber, "Long short-term memory," 1997.
[11] D. Hebb, *The Organization of Behavior.* New York: Wiley & Sons, 1949.
[12] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun, "Overfeat: Integrated recognition, localization and detection using convolutional networks," 2014.
[13] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation supplementary material," 2014.
[14] R. Girshick, "Fast r-cnn," 2015.
[15] S. Ren, K. He, R. Girshick, , and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," 2016.
[16] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015.
[17] K. Xu, J. L. Ba, R. Kiros, K. Cho, *et al.*, "Show, attend and tell: Neural image caption generation with visual attention," 2015.
[18] M. Jaderberg, K. Simonyan, A. Zisserman, and K. Kavukcuoglu, "Spatial transformer networks," 2016.
[19] N. Ballas, L. Yao, C. Pal, and A. Courville, "Delving deeper into convolution networks for learning video representation," 2016.