David J. Miller
Zhen Xiang
George Kesidis

Adversarial Learning and Secure AI

CAMBRIDGE
UNIVERSITY PRESS & ASSESSMENT

Chapter 02

Background on Deep Neural Networks

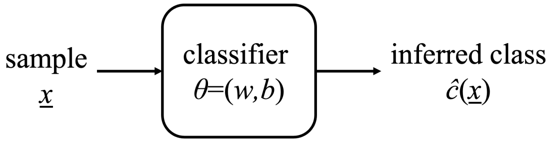
Outline

- Classification problems
- Motivating Deep (large) Neural Network (DNN) classifiers
- Neurons and DNN architectures
- Numerical training of DNNs (supervised deep learning)

Classification problems

- Consider many data samples in a large feature space.
- The samples may be, *e.g.*, images, documents, segments of speech, or may represent the current state of an online game.
- Suppose that, based on each sample, one of a finite number of decisions must be made.
- Plural samples may be associated with the same decision, *e.g.*,
  - the type of animal in an image,
  - the word that is being spoken in a segment of speech,
  - the sentiment or topic of some text, or
  - the action that is to be taken by a particular player at a particular state in the game.
- Thus, we can define a *class* of samples as all of those associated with the *same* ground-truth class label.

Classifier



- A sample x is an input pattern to a classifier $\hat{c}$.
- The output $\hat{c}(x)$ is the inferred class label (decision) for the sample x.
- The classifier parameters $\theta = (w, b)$ need to be learned so that the inferred class decisions are mostly accurate.

Types of data

- The samples themselves may have features that are of different types, *e.g.*, categorical, discrete numerical (ordinal), continuous numerical.
- There are different ways to transform data of all types to continuous numerical.
- How this is done may significantly affect classification performance.
- This is part of an often complex, initial data-preparation phase of deep learning (DNN training).
- In the following, we assume all samples $x \in \mathbb{R}^N$ for some (input) feature dimension N.

Training and test datasets for classification

- Consider a finite training dataset $\mathcal{T} \subset \mathbb{R}^N$ with **ground truth** class labels $c(x)$ for all $x \in \mathcal{T}$.
- $\mathcal{T}$ has representative samples of all K classes,

$$c : \mathcal{T} \to \{1, 2, ..., K\}.$$

- Using $\mathcal{T}, c$, the goal is to create a classifier

$$\hat{c} : \mathbb{R}^N \to \{1, 2, ..., K\} \text{ that}$$

  - accurately classifies on $\mathcal{T}$, *i.e.*, for most $x \in \mathcal{T}$, $\hat{c}(x) = c(x)$, and
  - hopefully generalizes well to an unlabelled production/test set $\mathcal{I}$ encountered in the field with the same distribution as $\mathcal{T}$,
  - *i.e.*, hopefully for most $x \in \mathcal{I}$, $\hat{c}(x) = c(x)$.

Training and test datasets for classification (cont)

- That is, the classifier "infers" the class label of the test samples $x \in \mathcal{I}$.
- To learn decision-making hyperparameters, a held-out subset of the training set, $\mathcal{H}$, with representatives from all classes, may be used to ascertain the *accuracy* of a classifier $\hat{c}$ on $\mathcal{H}$ as

$$\frac{\sum_{x \in \mathcal{H}} \mathbf{1}\{\hat{c}(x) = c(x)\}}{|\mathcal{H}|} \times 100\%.$$

Optimal Bayes error rate

- The test/production set $\mathcal{I}$ is *not* available or known (or is simply held out) during training.
- There may be some ambiguity when deciding about some samples.
- For each sample/input-pattern x, there is a *true* posterior distribution on the classes $p(\kappa|x)$, where $p(\kappa|x) \geq 0$ and $\sum_{\kappa=1}^{K} p(\kappa|x) = 1$.
- This gives the Bayes error (misclassification) rate, *e.g.*,

$$B \ := \ \int_{\mathbb{R}^N} (1 - p(c(x)|x))\psi(x)\mathrm{d}x,$$

where ψ is the (true) prior density on the input sample-space $\mathbb{R}^N$.

Optimal Bayes error rate[1]

- A given classifier $\hat{c}$ trained on a finite training dataset $\mathcal{T}$ (hopefully sampled according to ψ) may have normalized outputs for each class, $\hat{p}(\kappa|x) \geq 0$, *i.e.*, softmax output layers (described below) where

$$\hat{c}(x) = \arg\max_{\kappa} \hat{p}(\kappa|x).$$

- The classifier will have error rate

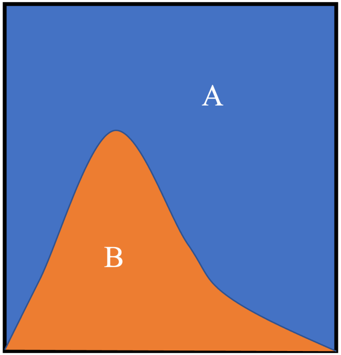
$$\int_{\mathbb{R}^N} \mathbf{1}\{\hat{c}(x) \neq c(x)\}\psi(x)\mathrm{d}x \ \geq \ B.$$

[1] See, *e.g.*, Duda, Hart and Stork. *Pattern Classification*. 2nd Ed. Wiley, 2001.

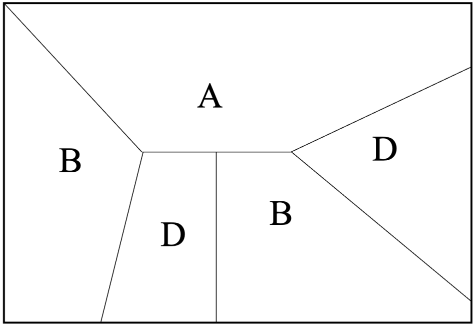
Motivating Deep (large) Neural Network (DNN) classifiers

- Consider a large training set $\mathcal{T} \subset \mathbb{R}^N$ ($|\mathcal{T}| \gg 1$) in a high-dimensional feature space ($N \gg 1$) with a possibly large number of associated classes ($K \gg 1$).
- In such cases, class decision boundaries may be nonconvex, and each class may consist of multiple disjoint regions (components) in feature space $\mathbb{R}^N$.
- So a highly parameterized classifier, *e.g.*, Deep artificial Neural Network (DNN), is warranted.
- Note: $A \subset \mathbb{R}^N$ is a convex set iff $\forall x, y \in A$ and $\forall r \in [0, 1]$, $rx + (1 - r)y \in A$.

Non-convex classes $\subset \mathbb{R}^2$



single-component classes
A & B are not convex



all components are convex,
A is convex, B & D are not

Cover's theorem

Theorem: If $K = 2$ and the two classes represented in $\mathcal{T} \subset \mathbb{R}^N$ are not linearly separable, then there is a continuous mapping μ such that they are linearly separable in

$$\mu(\mathcal{T}) = \{\mu(x) \mid x \in \mathcal{T}\}.$$

Proof:

- Choose an enumeration $\mathcal{T} = \{x^{(1)}, x^{(2)}, ..., x^{(T)}\}$ where $T = |\mathcal{T}|$.
- Continuously map each sample x to a different unit vector $\in \mathbb{R}^T$;
- *i.e.*, $\forall k$, $\mu(x^{(k)}) = \mathrm{e}^{(k)}$, where $\mathrm{e}_k^{(k)} = 1$ & $\mathrm{e}_j^{(k)} = 0 \ \forall j \neq k$.

Proof of Cover's theorem (cont)

- For example, use Lagrange interpolating polynomials with 2-norm $\|\cdot\|$ in $\mathbb{R}^N$:

$$\forall k, \ \mu_k(x) = \prod_{j=1, j\neq k}^{T} \frac{\|x - x^{(j)}\|}{\|x^{(k)} - x^{(j)}\|},$$

- where $\mu = [\mu_1, ..., \mu_T]' : \mathbb{R}^N \to \mathbb{R}^T$, and
- v' is the transpose of vector v, so that the inner product (dot product)

$$\langle v, u \rangle = v'u.$$

Proof of Cover's theorem (cont)

- Every partition of the samples $\mu(\mathcal{T}) = \{\mu(x) \mid x \in \mathcal{T}\}$ into two different index-sets (classes) κ_1 and κ_2 is separable by the hyperplane with parameters

$$w = \sum_{k\in\kappa_1} \mathrm{e}^{(k)} - \sum_{k\in\kappa_2} \mathrm{e}^{(k)} \quad \text{(so } w \in \mathbb{R}^T \text{ has entries } \pm 1\text{).}$$

- Thus, $\forall k \in \kappa_1,\ w'\mathrm{e}^{(k)} = 1 > 0$, and $\forall k \in \kappa_2,\ w'\mathrm{e}^{(k)} = -1 < 0$.
- Q.E.D.
- We can build a classifier for $K > 2$ classes from K such linear, binary classifiers, *e.g.*,
  - Consider class partition $\mathcal{T}_1, \mathcal{T}_2, ..., \mathcal{T}_K$ of $\mu(\mathcal{T})$.
  - i^{th} binary classifier separates $\mathcal{T}_i$ from $\cup_{j\neq i}\mathcal{T}_j$, *i.e.*, "one versus rest".

Cover's theorem - Remarks

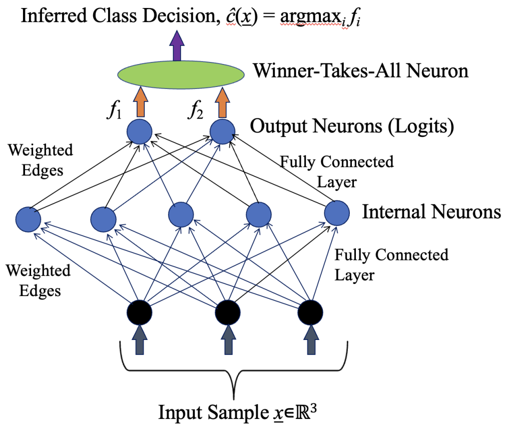
- Here, $\mu(x)$ is analogous to DNN mapping from input x to an internal layer.
- If the feature dimension is already much larger than the number of samples (*i.e.*, $N \gg K$ as in, *e.g.*, some genome datasets), then the data $\mathcal{T}$ will likely already be linearly separable.
- According to Cover's theorem in this case, there is a mapping to a *lower* dimensional space in which the data is linearly separable.

DNN architectures

Outline:

- Some types of neurons/units (activation functions)
- Some types of layers
- Example DNN architectures especially for image classification

Illustrative 2-class feed-forward neural network



Some types of neurons

- Consider a neuron/unit n in layer $\ell(n)$, *i.e.*, $n \in \ell(n)$, with input edge-weights $w_{i,n}$, where neurons i are in layer prior (closer to the input) to that of n, *i.e.*, $i \in \ell_-(n)$.
- The activation of neuron n is (for the example of a fully connected layer)

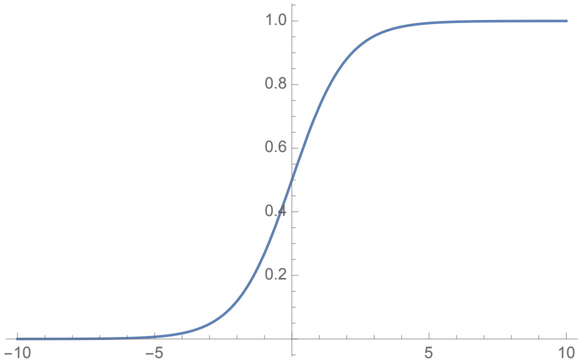
$$v_n = f\left(\sum_{i \in \ell_-(n)} v_i w_{i,n}\ ,\ \ b_n\right),$$

  where b_n are additional parameters of the activation itself.
- Neurons of the linear type have activation functions of the form

$$f(z, b_n) = b_{n,1} z + b_{n,0},$$

  where slope $b_{n,1} > 0$ and $b_{n,0}$ is a "bias" parameter.

Sigmoid activation function



Some types of neurons (cont)

- Neurons of the sigmoid type have activation functions that include

$$f(z, b_n) = \tanh(zb_{n,1} + b_{n,0}) \in (-1, 1), \quad \text{or}$$

$$f(z, b_n) = \frac{1}{1 + \exp(-zb_{n,1} - b_{n,0})} \in (0, 1), \quad \text{where} \quad b_{n,1} > 0.$$

- Rectified Linear Unit (ReLU) type activation functions include

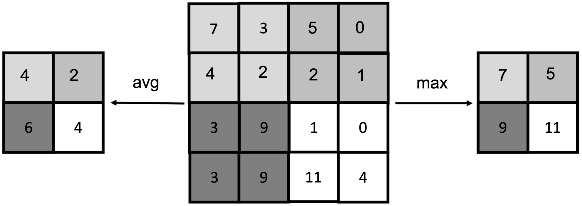
$$f(z, b_n) = (b_{n,1}z + b_{n,0})^+ = \max\{b_{n,1}z + b_{n,0},\ 0\}.$$

- Note that ReLUs are not continuously differentiable at $z = -b_{n,0}/b_{n,1}$.
- Also, both linear and ReLU activations are unbounded, whereas sigmoids are bounded.
- ReLU parameters are often not learned and just taken to be $b_{n,0} = 0$, $b_{n,1} = 1$.

Some types of layers - fully connected

- Consider neurons n in layer $\ell = \ell(n)$.
- If it's possible that $w_{i,n} \neq 0$ for all $i \in \ell_-(n), n \in \ell$, then layer ℓ is said to be **fully interconnected**.

Pooling layers for downsampling



- Pooling layers are intended to downsample from a large layer ℓ_- to a smaller one ℓ, *i.e.*, $|\ell_-| \gg |\ell|$.
- Each number for the example above is a neural activation of a pooling layer, where
- $|\ell_-| = 16$ (center), $|\ell| = 4$, and
- the 2×2 window *slides* across the larger representation (ℓ_-) according to the *stride* parameter (2) to take $|\ell| = 4$ different *maximum* readings and form the downsampled *max*-pooling layer (ℓ at right).
- Or downsampling by, *e.g.*, *average* pooling (ℓ at left).

Convolutional layers

- Consider the activation of the n^{th} neuron of layer ℓ, $v_{\ell,n}$.
- Layer ℓ is said to be **convolutional** if, *e.g.*,

$$v_{\ell,n} = \sum_{i=1}^{m} w_{m-i+1} v_{\ell_-,n+i-1},$$

  where vector w is the convolution kernel or filter of size m and $1 \leq n \leq |\ell_-| - m + 1$.
- So, different neural activations in layer ℓ are dot products between the kernel w and different overlapping windows of layer ℓ_- activations,
- *i.e.*, the windows sweep across the neurons of ℓ_-.
- For images, 2D or 3D kernels may be used (3D when the image has multiple color channels per pixel).
- A kernel w can be trained to detect localized, class-discriminatory patterns positioned somewhere in the activations of ℓ_-.

Convolutional layers (cont)

- A single convolutional layer could use $W > 1$ different kernels $w^{(k)}$, $k \in \{1, 2, ..., W\}$, where all kernels are typically of the same size.
- Compared to fully connected layers with $|\ell_-| \cdot |\ell|$ parameters, convolutional layers have only $\sum_{k=1}^{W} |w^{(k)}|$ parameters.
- Convolutions are characteristic of linear, shift-invariant transformations, which were used for decades in data processing prior to their incorporation into neural networks, and continue to be used today.

Nearest-prototype final layer

- Assuming a penultimate layer with activations $z \in \mathbb{R}^M$ for input sample s, the idea is to learn a prototype $b_\kappa \in \mathbb{R}^M$ for each class $\kappa \in \{1, 2, .., K\}$.
- The final-layer activations are, *e.g.*,

$$f_\kappa(z) = \phi(\|z - b_\kappa\|_2^2)$$

  where ϕ is a smooth, positive, increasing function with $\phi(0) = 0$.
- The use of Euclidean Radial Basis Functions (RBFs), *i.e.*, $\phi(x) \equiv x$, in this layer is equivalent to a K-component Gaussian Mixture Model (GMM) with identity covariances and hard assignments to components.
- For a final **nearest-prototype layer**, the class decision is the *minimum* of the final layer activations,

$$\hat{c} = \arg\min_{\kappa} f_\kappa.$$

Softmax class decisions based on the final layer

- Again, suppose the DNN has K outputs f_κ for classes $\kappa \in \{1, 2, \ldots, K\}$.
- If $f_\kappa(x) \geq 0$ for all (DNN inputs) x, then we may define, *e.g.*,

$$\hat{p}(\kappa|x) \quad = \quad f_\kappa(x) \Big/ \sum_{j=1}^{K} f_j(x) \ ,$$

  else we may define, *e.g.*,

$$\hat{p}(\kappa|s) \quad = \quad \exp(bf_\kappa(x)) \Big/ \sum_{j=1}^{K} \exp(bf_j(x)) \quad \text{with } b > 0.$$

- These terms may approximate the true posterior class probabilities and form the output of a *softmax* layer.
- For each input x, a *winner take all* output layer gives the class decision

$$\hat{c}(x) = \arg\max_\kappa \hat{p}(\kappa|x) = \arg\max_\kappa f_\kappa(x).$$

Softmax layer - classification confidence

- Classification "confidence" can be defined as

$$\frac{\hat{p}(\hat{c}(x)|x) - \max_{i \neq \hat{c}(x)} \hat{p}(i|x)}{\hat{p}(\hat{c}(x)|x)} \in [0,1].$$

- The class decision for x may *not be accepted* unless it has some "margin" $\mu_\kappa > 0$, *i.e.*, unless

$$\frac{\hat{p}(\hat{c}(x)|x) - \max_{i \neq \hat{c}(x)} \hat{p}(i|x)}{\hat{p}(\hat{c}(x)|x)} > \mu_{\hat{c}(x)}.$$

- The parameters μ_κ could be set using a labelled validation set $\mathcal{H}$ which is held out of training.

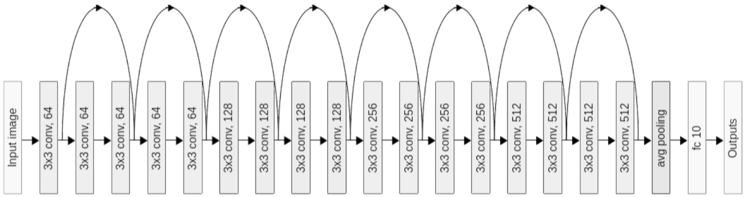
Batch normalization layer

- Batch normalization layers are used in ResNet, see examples below.
- May be needed because ReLU neurons have positively unbounded output signals.
- For example, in a batch normalization layer, the output activations of the i^{th} neuron whose input is x_i is given by

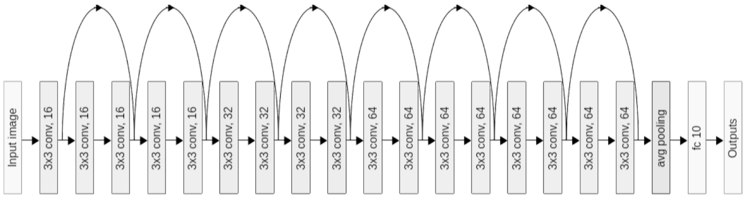
$$\frac{x_i - \mu_i}{\sigma_i}\gamma + \beta.$$

- The two layer-parameters γ, β are learned.
- For each neuron, the mean and variance estimates μ_i, σ_i^2 are dynamically updated, using the current training batch (see SGD below), by a simple first-order autoregressive mechanism with forgetting factor set as, *e.g.*, 0.9.

ResNet wide architecture



ResNet compact architecture



DNN architectures - Discussion

- The “front end” performs abstract feature extraction, *e.g.*, convoluational layers.
- The “back end” makes class decisions based on combinations of abstracted features, *e.g.*, fully connected layers.
- That is, the class decision boundaries in the input (raw) feature space are much more complex than those based on the activations of the final layer of the DNN front end.
- The final softmax layer allows for assessment of class-decision confidence.
- There are other types of neurons (*e.g.*, gated), layers (*e.g.*, attention), and architectures (*e.g.*, recurrent, transformer), for other types of applications (*e.g.*, generative modeling, regression/prediction, translation).

Optimization methods for training

Outline:

- Background on gradient based optimization methods
- Types of training/learning objectives
- Stochastic Gradient Descent (SGD) with momentum
- Background on first-order autoregressive (AR-1) estimators
- Overfitting and regularization
- Training dataset augmentation
- Held-out validation set for hyperparameters

Background on gradient based methods[2]

Outline:

- Directional derivative and descent directions
- First and second order optimality conditions
- Gradient methods for local optimality

[2] See: E. Polak. *Notes on Fundamentals of Optimization For Engineers.* Spring, 1990. https://www2.eecs.berkeley.edu/Pubs/TechRpts/1989/ERL-89-40.pdf

Gradients of continuously differentiable functions

- Consider a continuously differentiable function $L : \mathbb{R}^n \to \mathbb{R}$ for integer $n \geq 1$.
- Our objective is to find a local minimum $\hat{x} \in \mathbb{R}^n$ of L.
- The gradient of L is

$$\nabla L(x) = \begin{bmatrix} \frac{\partial L}{\partial x_1}(x) \\ \frac{\partial L}{\partial x_2}(x) \\ \vdots \\ \frac{\partial L}{\partial x_n}(x) \end{bmatrix}.$$

- Note that $\nabla L : \mathbb{R}^n \to \mathbb{R}^n$.

Directional derivatives

- The directional derivative of L at x in the direction h is
$$(\nabla L(x))' h = \langle \nabla L(x), h \rangle = \lim_{\eta \to 0} \frac{L(x+\eta h) - L(x)}{\eta}.$$
- Here, scalar $\eta \in \mathbb{R}$, and vectors $x, h \in \mathbb{R}^n$.
- h is a descent direction of L at x if $\langle \nabla L(x), h \rangle \ < 0$.
- Obviously, $-\nabla L(x)$ is a descent direction at x unless $\nabla L(x) = 0$.
- **Theorem:** If h is a descent direction of L, then there is a $\eta > 0$ such that
  $L(x+\eta h) < L(x)$.
- **Proof:** By the previous display, there is a sufficiently small $\eta > 0$ such that
$$\begin{aligned} \frac{L(x+\eta h) - L(x)}{\eta} - \langle \nabla L(x), h \rangle \quad &\leq \quad -\frac{1}{2} \langle \nabla L(x), h \rangle \\ \Rightarrow \quad L(x+\eta h) - L(x) \quad &\leq \quad \frac{\eta}{2} \langle \nabla L(x), h \rangle \ < \ 0. \end{aligned}$$

Optimality conditions - necessity

- $\hat{x}$ is a local minimum of L if there is a $r > 0$ such that $L(\hat{x}) \leq L(x)$ for all $x \in \mathcal{B}(\hat{x}, r) = \{y : \|y - \hat{x}\|_2 < r\}$ (open ball centered at $\hat{x}$ with radius r).
- **Theorem:** If $\hat{x}$ is a local minimizer of L then $\nabla L(\hat{x}) = 0$.
- **Proof:** Assume $\nabla L(\hat{x}) \neq 0$, use the descent direction $h = -\nabla L(\hat{x})$ and argue as previous theorem (with $\eta < r$) to contradict local minimality of $\hat{x}$.
- The Hessian of (twice continuously differentiable) L is the $n \times n$ matrix

$$H = \frac{\partial^2 L}{\partial x^2} = \left[\frac{\partial^2 L}{\partial x_i \partial x_j}\right]_{i,j=1}^{n}$$

- Note that $H : \mathbb{R}^n \to \mathbb{R}^{n \times n}$.

Optimality conditions - necessity (cont)

Theorem: If $\hat{x}$ is a local minimizer of L, then $\forall h$, $\langle h, H(\hat{x})h\rangle \geq 0$, *i.e.*, $H(\hat{x})$ is positive semi-definite.

Proof:

- For $x, y \in \mathbb{R}^n$, $s \in [0,1]$, let $g(s) = L(x + s(y-x))$.
- Integrating $g''(s)(1-s) = (g'(s)(1-s) + g(s))'$ gives

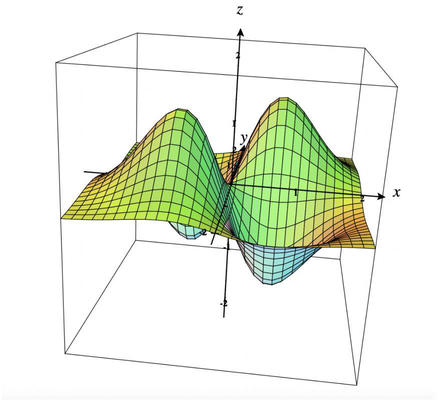
$$L(y) - L(x) \quad = \quad \langle \nabla L(x), y - x\rangle + \int_0^1 (1-s)\langle y - x, H(x + s(y - x$$

- Substitute $y = \hat{x} + \eta h$, $x = \hat{x}$, and $\nabla L(\hat{x}) = 0$.
- Finally, let $\eta \to 0$.

Optimality conditions - sufficiency

- **Theorem:** If $\nabla L(\hat{x}) = 0$ and, $\forall h$, $\langle h, H(\hat{x})h \rangle > 0$, then $\hat{x}$ is a local minimizer.
- To prove this, assume $\hat{x}$ is not a local miminizer.
  - So there is a sequence $x_i \to \hat{x}$ such that $L(x_i) < L(\hat{x})$ for all $i \in \mathbb{N}$.
  - Then argue as the previous theorems to show a contradiction with the hypothesis.
- For $n = 1$, recall that if $L'(\hat{x}) = 0$ and $L''(\hat{x}) > 0$ then $\hat{x}$ is a local minimum of L.

Local minima, maxima and (if $n > 1$) saddle points



Plot of $z = 7xye^{-x^2-y^2}$ from `https://c3d.libretexts.org/CalcPlot3D/index.html`

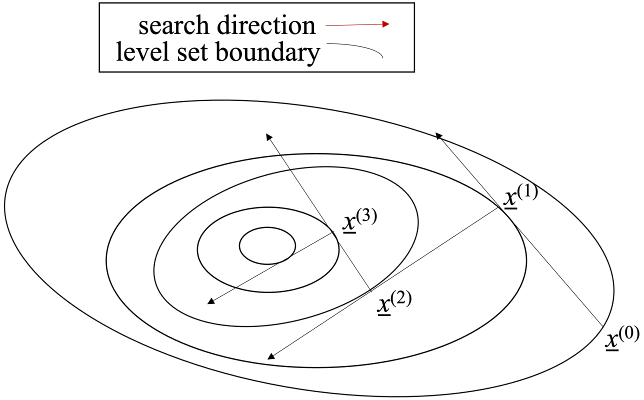
- two local minima: $\nabla L = 0$ and H is positive definite
- two local maxima: $\nabla L = 0$ and H is negative definite
- one saddle at origin: $\nabla L = 0$ and H is indefinite (dim $n > 1$)

Steepest Descent algorithm

1. initially, $x^{(0)} \in \mathbb{R}^n$, iteration index $k = 0$, small $\varepsilon > 0$
2. if $\|\nabla L(x^{(k)})\| < \varepsilon$ then stop
3. search (descent) direction $h^{(k)} = -\nabla L(x^{(k)})$
4. line search to find step size:
$$\eta^* = \arg\min_{\eta > 0} L(x^{(k)} + \eta h^{(k)})$$
5. update $x^{(k+1)} = x^{(k)} + \eta^* h^{(k)}$
6. k++ and go to step 2.

Note that determining η^* is a one-dimensional optimization problem.

Line search terminates at point where $-\eta \nabla L(x^{(k)})$ is tangent to a level-set of L



Steepest descent - convergence

- Can show by contradiction that any accumulation point $\hat{x}$ (limit of a convergent subsequence) of the sequence $x^{(k)}$ must satisfy $\nabla L(\hat{x}) = 0$.
- Thus, if $H(\hat{x})$ is positive definite (so $\hat{x}$ is not a saddle) then $\hat{x}$ is a local minimum.
- Additional Wolfe condition on curvature of L guarantees convergence of gradient descent to a local minimum: $\exists c > 0$ s.t. $\forall k$,

$$|\langle h^{(k)}, \nabla L(x^{(k)} + \eta_k h^{(k)}) \rangle| \ \leq \ c|\langle h^{(k)}, \nabla L(x^{(k)}) \rangle|.$$

Optimization heuristics for Deep Learning[3]

- Approaches that leverage second-order derivatives (Newton-Raphson) or their approximations (BFGS or DFP quasi-Newton methods) are too complex for deep learning.
- There are simpler approaches to line search (Armijo), which may still be too complex for the DNN setting.
- In the following, we describe some heuristics that are used to train DNNs.

[3] See: S. Ruder. An overview of gradient descent optimization algorithms. https://ruder.io/optimizing-gradient-descent/, 19 Jan 2018.

Types of training objectives

- Learning objective is to choose classifier parameters $\theta = (w, b)$, *i.e.*, *train* the classifier, to **minimize** the following "loss" expressions over the DNN parameters θ **on which final-layer activations f_κ, $\kappa \in \{1, 2, ..., K\}$ and inferred class decisions** $\hat{c} = \arg\max_\kappa f_\kappa$, **implicitly depend.**
- The training-set misclassification-rate objective,

$$L = \frac{1}{|\mathcal{T}|} \sum_{x \in \mathcal{T}} \mathbf{1}\{\hat{c}(x) \neq c(x)\},$$

is not differentiable (w.r.t. θ), and so would not lend itself to training by gradient based methods.

MSE loss objective

- A Mean Square Error (MSE) loss objective is,

$$L = \frac{1}{|\mathcal{T}|} \sum_{x \in \mathcal{T}} |\hat{c}(x) - c(x)|^2.$$

- Note how MSE depends on the numerical values assigned to labels while the misclassification rate is based on categorical class labels.
- MSE is often used for regression or prediction applications.

Cross-entropy loss objective

- A cross-entropy loss objective is, *e.g.*,

$$L = -\frac{1}{|\mathcal{T}|}\sum_{x\in\mathcal{T}} \log \hat{p}(c(x)|x) \geq 0,$$

  where the DNN's class posterior

$$\hat{p}(\kappa|x) = \frac{f_\kappa(x)}{\sum_j f_j(x)}$$

  and logit activations $f_j \geq 0$ are differentiable w.r.t. DNN parameters θ.
- To simplify for deep learning, we can replace $\hat{p}(c(x)|x)$ by $f_{c(x)}(x)$ in the expression for L.
- Cross-entropy loss objectives are commonly used and optimized using gradient-based methods.
- Note that the class labels c are categorical here.

Back propagation to compute gradients of feed-forward neural networks

- Back propagation is just the chain rule for differentiation to compute the gradient of composed functions.
- Consider a function of two real variables $g(\theta_1, \theta_2)$ and define

$$\partial_1 g = \frac{\partial g_1}{\partial \theta_1} \text{ and } \partial_2 g = \frac{\partial g_1}{\partial \theta_2}.$$

- Now consider the following composed function of four variables

$$L(x, \theta_1, \theta_2, \theta_3) = g_3(\theta_3, g_2(\theta_2, g_1(\theta_1, x)))$$

  where g_3, g_2, g_1 are functions of two variables.
- L represents a loss function of a simple, feed-forward neural network consisting of just three consecutive neurons (one per layer) having differentiable activations g;
- here, θ_k is the parameter associated with layer k, g_k is the output of layer k, and the DNN output layer is indexed 3.

Back propagation (cont)

- Again, $L(x, \theta_1, \theta_2, \theta_3) = g_3(\theta_3, g_2(\theta_2, g_1(\theta_1, x)))$.
- Simply by the chain rule, the gradient of L is

$$\nabla_\theta L = \begin{bmatrix} \partial L/\partial\theta_3 \\ \partial L/\partial\theta_2 \\ \partial L/\partial\theta_1 \end{bmatrix} = \begin{bmatrix} \partial_1 g_3 \\ \partial_2 g_3 \cdot \partial_1 g_2 \\ \partial_2 g_3 \cdot \partial_2 g_2 \cdot \partial_1 g_1 \end{bmatrix}$$

- Note that to compute $\partial L/\partial\theta_k$ for $k = 1, 2$, one needs to compute $\partial_2 g_3$, *i.e.*, this quantity needs to be propagated back from layer 3.
- In a similar way for a more complex feed-forward neural network, to compute the partial derivative of a loss function with respect to parameters in layer ℓ of a DNN, the partial derivatives with respect to parameters from layers closer to the output need to be **propagated back** to layer ℓ.
- Also note that computing $\partial_2 g_3$ requires the layer-2 activation by input x, $g_2(\theta_2, g_1(\theta_1, x))$.

Vanishing gradient problem

- The gradients computed using back propagation may become very small in magnitude as the number of layers increases in some neural network designs.
- ResNet's use of "forward branching" mitigates this vanishing gradient effect.
- Recurrent neural networks which use inter-sample neural memory and "back propagation through time" also may avoid vanishing gradient problems.

Constant learning rate instead of optimal step size

- Rather than attempting to compute an optimal step size per iteration of gradient descent, one can simply take a constant step size, $\eta > 0$, *i.e.*, $x_k = x_{k-1} + \eta h^{(k)}$.
- Here, $\eta > 0$ is also called the *learning rate*.
- Typically, chosen learning rate $\eta \in [0.01 - 0.99]$, *e.g.*, $\eta = 0.1$.
- This said, η may change dynamically, particularly η becomes smaller as iteration index k increases for greater "depth" of search,
- as opposed to greater "breadth" with larger η initially (when k is small),
- *e.g.*, η can be reduced by a factor of 10 every 10 iterations of gradient descent.

Stochastic Gradient Descent (SGD)

- Suppose an *additive* loss objective to be minimized

$$L(\theta) = \frac{1}{T}\sum_{j=1}^{T} g_j(\theta)$$

where $g_j(\theta) = g(\theta, x^{(j)})$ for $x^{(j)} \in \mathcal{T}$, $T = |\mathcal{T}|$.

- When $T \gg 1$, computing $\nabla_\theta g_j$ for all j can be very costly.
- Instead consider an equal-partition of $\mathcal{T}$, $B_0, B_1, \ldots, B_{M-1}$, and at step k use search direction

$$h^{(k)} = -\frac{1}{|B_{k \mod M}|}\sum_{j\in B_k} \nabla g_j(\theta).$$

where $|B_k| = T/M \ll T$.

- Note that $h^{(k)}$ for a given k may not be a descent direction for L.

Review of first-order autoregressive estimators

- Suppose we want to iteratively estimate the mean of the possibly nonstationary sequence $\overline{X}_n$, for $n \in \{0, 1, 2, ...\}$, and possibly with an unknown (stationary) limiting distribution.
- Since the distribution of X_n may change with n, one could want to more significantly weight the recent samples X_k (*i.e.*, $k \leq n$ and $k \approx n$) in the computation of $\overline{X}_n$.
- An order-1 autoregressive estimator (AR-1) is

$$\overline{X}_n = \alpha \overline{X}_{n-1} + (1 - \alpha) X_n$$

where $0 < \alpha < 1$ is the forgetting/fading factor and $\overline{X}_0 = X_0$.

AR-1 estimators (cont)

- All past values of X contribute to the current value of $\overline{X}$ according to weights that exponentially diminish:

$$\overline{X}_n = \alpha^n X_0 + (1-\alpha)(\alpha^{n-1}X_1 + \alpha^{n-2}X_2 + \ldots + \alpha X_{n-1} + X_n).$$

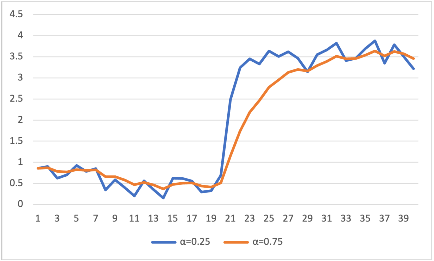
- Also, if $1-\alpha$ is a power of 2, then the autoregressive update

$$\overline{X}_n = \overline{X}_{n-1} + (1-\alpha)(X_n - \overline{X}_{n-1}),$$

  is simply implemented with two additive operations and one bit-shift (the latter to multiply by $1-\alpha$).
- There is a simple trade-off in the choice of α.
- A small α implies that $\overline{X}_n$ is more responsive to the recent samples X_k ($k < n, k \approx n$), but this can lead to undesirable oscillations in the AR-1 process $\overline{X}$.
- A large value of α means that $\overline{X}$ will have diminished oscillations ("low-pass" filter) but will be less responsive to changes in the distribution of the samples X.

AR-1 estimators - example

- Suppose the initial distribution is uniform on the interval $[0, 1]$ (*i.e.*, $\mathsf{E}X = 0.5$), but for $n \geq 20$ the distribution is uniform on the interval $[3, 4]$ (*i.e.*, $\mathsf{E}X$ changes to 3.5).
- When $\alpha = 0.2$, a sample path of the first-order AR-1 process $\overline{X}$ responds much more quickly to the change in mean (at $n = 20$), but is more oscillatory than the corresponding sample path of the AR-1 process when $\alpha = 0.8$.



SGD with momentum[4]

- Momentum incorporates information from **prior** "stochastic gradients" $h^{(j)}$ for $j < k$ to try to improve possibly crude approximations $h^{(k)}$ of $-\nabla L$.
- For example, using simple first-order autoregression with forgetting/fading factor $\alpha \in (0,1)$, take search direction

$$H^{(k)} = \alpha H^{(k-1)} + (1-\alpha) h^{(k)}$$

  where H_{k-1} is the search direction used for the previous set of DNN parameters, $\theta^{(k-1)}$.
- Thus, $\theta^{(k)} = \theta^{(k-1)} + \eta H^{(k)}$.
- To further simplify in this land of heuristics, take

$$\theta^{(k)} = \theta^{(k-1)} + H^{(k)} \quad \text{with} \quad H^{(k)} = \alpha H^{(k-1)} + \eta h^{(k)}.$$

[4] See: https://arxiv.org/pdf/1609.04747.pdf

SGD with momentum (cont)

- SGD's "randomness" and momentum may avoid zigzagging through "ravines" associated with shallow local minima of L,
- where zigzag is indicated by persistently negative sign of $\langle h^{(k-1)}, h^{(k)} \rangle$.
- Typically, chosen momentum parameter $\alpha \in [0.1, 0.9]$, *e.g.*, $\alpha = 0.8$.
- The commonly used "Adam" optimizer and RMS techniques normalize an autoregressive estimate of the gradient by an autoregressive estimate of its (uncentered) second moment.

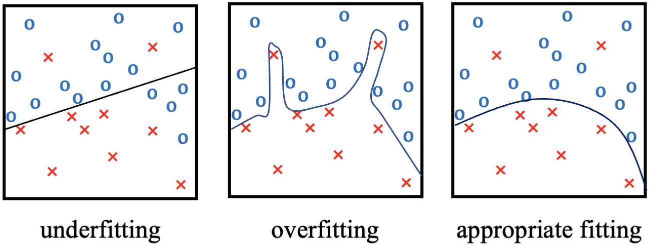
Overfitting and DNN regularization

- "We may assume the superiority … of the demonstration which derives from fewer postulates or hypotheses." Aristotle, *Posterior Analytics* (as Occam's Razor).
- That is, best generalization performance if a minimum number parameters is used to explain the training data, *i.e.*, avoid overfitting to the training set $\mathcal{T}$.
- Note that *a priori* no idea how many parameters are suitable for very complex training datasets (large number of samples in large feature dimensions), and the number of DNN parameters can be very large.
- So DNN may or may not be (initially) overparameterized.

Overfitting and DNN regularization (cont)

- Low accuracy on the training set may indicate too few parameters (insufficient DNN capacity to learn),
- while low accuracy on the held-out validation set (poor generalization performance) with high accuracy on the training set may indicate too many parameters (overfitting to the training set).
- Can heuristically reduce the number parameters, *e.g.*, by removing parameters which are close to zero during training, while checking performance on a held-out validation set.

Overfitting illustrated[5]



- “Appropriate” fitting has better generalization performance on the test set by not overfitting to the two “X” class outliers.
- A held-out evaluation set $\mathcal{H}$ can be used to navigate between underfitting and overfitting.

[5] See: Shubham Jain. An Overview of Regularization Techniques in Deep Learning, `https://www.analyticsvidhya.com/blog/2018/04/fundamentals-deep-learning-regularization-techniques/`, April 19, 2018.

Random Dropout

- Under “dropout,” only a randomly selected fraction of DNN parameters are updated at each iteration of gradient-based deep learning.
- So at each iteration of gradient based deep learning using dropout in combination with SGD, only subset of training samples are used to update only a subset of the network parameters θ.
- Presuming that the network is over-parameterized, this has the “regularizing” effect of “spreading” the learning throughout the network, so overfitting is potentially avoided.

Training dataset augmentation

- Consider the training dataset of an image classifier including images that belong to say the *cat* class.
- To improve generalization performance on the test/production set, the training set can be augmented with a version of each cat image that is rotated, cropped, tint/color adjusted, contrast adjusted, has noise added, *etc.*
- But in some cases, augmenting with samples that are close to training samples (*e.g.*, augmenting with "adversarial" samples in an attempt to be robust to test-time evasion attacks) may cause overfitting to training samples and degrade generalization performance.

Held-out validation set for hyperparameters

- Some training samples used to learn "main" classifier parameters $\theta = (w, b)$, while a (uniformly) sampled held-out validation $\mathcal{H}$ set is used to tune, *e.g.*,
  - training hyperparameters, *e.g.*, initial weights, learning rate, forgetting factors, bounds on or normalizations of classifier parameters,
  - parameters for representing and preprocessing data (before training), or
  - parameters of the DNN architecture (number, types and sizes of layers).
- Also, can simply "early stop" (terminate) training when accuracy degrades on $\mathcal{H}$ (or instead on $\mathcal{T}$) to prevent overfitting.
- Again, the validation set is uniformly sampled from the training set so that it is unbiased.
- The validation set $\mathcal{H}$ is *not* the unlabelled production/test set $\mathcal{I}$, and *only the rest* of the training set ($\mathcal{T}$) is used to learn θ.

Concluding comments

- DNNs have achieved state-of-the-art performance in some important application domains.
- DNNs and the datasets they classify are often very large-scale.
- DNNs have highly heterogeneous architectures and are highly nonlinear; class partitions of "raw" input feature space are highly nonconvex.
- In practice, optimization mechanisms used, and neural and network-architectural choices made, are heuristic, trial-and-error affairs[6], when they are not based on classical ideas (*e.g.*, regression, convolutions, AR-1, gradient descent, residual signals).
- Data representation, formatting and curating to produce $\mathcal{T}$ and $\mathcal{H}$, requiring actual domain expertise, may be much more time-consuming and costly than DNN training/inference.[7]

[6] S. Higginbotham. Show Your Machine-Learning Work. *IEEE Spectrum*, Dec. 2019.

[7] *e.g.*, E. Strickland. How IBM Watson Overpromised and Underdelivered on AI Health Care. *IEEE Spectrum*, Apr. 2019; J. Murdock Google's AI Health Screening Tool Claimed 90 Percent Accuracy, but Failed to Deliver in Real World Tests. *Newsweek*, 4/28/20.

With Permission, Figures Reproduced From

- Z. Xiang, D.J. Miller, and G. Kesidis. Reverse engineering imperceptible backdoor attacks on deep neural networks for detection and training set cleansing. *Computers and Security* **106**, July 2021.