Concise Introduction to Deep Neural Networks

Outline:

• Classification problems

• Motivating Deep (large) Neural Network (DNN) classifiers

• Neurons and DNN architectures

• Numerical training of DNNs (supervised deep learning)

• Spiking and gated neurons

• Concluding remarks

Glossary

$N$: dimension of sample (classifier input pattern) space, $\mathbb{R}^N$

$\mathcal{T}$: finite set of labelled training samples $s \in \mathbb{R}^N$, i.e., $\mathcal{T} \subset \mathbb{R}^N$

$C$: the (finite) number of classes

$c(s) \in \{1, 2, \ldots, C\}$: true class label of $s \in \mathbb{R}^N$.

$\mathcal{I}$: finite set of unlabelled test/production data samples $s \in \mathbb{R}^N$ to perform class inference, $\mathcal{I} \subset \mathbb{R}^N$

$\hat{c}(s)$: inferred class of sample $s$ by the neural network

$w$: edge weights of the neural network

$b$: neuron (or "unit") parameters

$x = (w, b)$: collective parameters of the neural network

$v$: neuron output (activation)

$f, g$: neuron activation function

$\ell$: a set of neurons comprising a network layer

$\ell^{n-1}$: a set of neurons comprising a network layer prior to that in which neuron $n$ resides

$L$: loss function used for training

$\eta$: learning rate or step size

$\alpha, \phi$: gradient momentum parameter, forgetting/fading factor

$\lambda$: Lagrange multiplier
Classification problems

- Consider many data samples in a large feature space.
- The samples may be, e.g., images, segments of speech, documents, or the current state of an online game.
- Suppose that, based on each sample, one of a finite number of decision 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 decision.

Classifier

- A sample $s$ is an input pattern to a classifier.
- The output $\hat{c}(s)$ is the inferred class label (decision) for the sample $s$.
- The classifier parameters $x = (w, b)$ 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, 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 $s \in \mathbb{R}^N$ for some 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 $c(s)$ for all $s \in \mathcal{T}$.
- $\mathcal{T}$ has representative samples of all $C$ classes,
  $$c : \mathcal{T} \to \{1, 2, ..., C\}.$$
- Using $\mathcal{T}, c$, the goal is to create a classifier
  $$\hat{c} : \mathbb{R}^N \to \{1, 2, ..., C\}$$
  that
  - accurately classifies on $\mathcal{T}$, i.e., for most $s \in \mathcal{T}$, $\hat{c}(s) = c(s)$, 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 $s \in \mathcal{I}$, $\hat{c}(s) = c(s)$.
- That is, the classifier “infers” the class label of the test samples $s \in \mathcal{I}$.
- 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
  $$\frac{\sum_{s \in \mathcal{H}} 1\{\hat{c}(s) = c(s)\}}{|\mathcal{H}|} \times 100\%.$$
Optimal Bayes error rate

- The test/production set $I$ is not available or known during training.
- There may be some ambiguity when deciding about some samples.
- For each sample/input-pattern $s$, there is a true posterior distribution on the classes $p(\kappa|s)$, where $p(\kappa|s) \geq 0$ and $\sum_{\kappa=1}^{C} p(\kappa|s) = 1$.
- This gives the Bayes error (misclassification) rate, e.g.,
  \[ B := \int_{\mathbb{R}^N} (1 - p(c(s)|s)) \psi(s) ds, \]
  where $\psi$ is the (true) prior density on the input sample-space $\mathbb{R}^N$.
- A given classifier $\hat{c}$ trained on a finite training dataset $T$ (hopefully sampled according to $\psi$) may have normalized outputs for each class, $\hat{p}(\kappa|s) \geq 0$, cf. softmax output layers where
  \[ \hat{c}(s) = \arg\max_{\kappa} \hat{p}(\kappa|s). \]
- The classifier will have error rate
  \[ \int_{\mathbb{R}^N} 1\{\hat{c}(s) \neq c(s)\} \psi(s) ds \geq B. \]

Motivating Deep (large) Neural Network (DNN) classifiers

- Consider a large training set $T \subset \mathbb{R}^N (|T| \gg 1)$ in a high-dimensional feature space ($N \gg 1$) with a possibly large number of associated classes ($C \gg 1$).
- 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 (large) artificial Neural Network (DNN), is warranted.
- Note: $A \subset \mathbb{R}^N$ is a convex set iff $\forall x, y \in A$ and $\forall r \in [0, 1]$, $rx + (1 - r)y \in A$. 

© Jan. 2020 George Kesidis
Non-convex classes $\subset \mathbb{R}^2$

Some alternative classification frameworks:

- Gaussian Mixture Models (GMMs) with BIC training objective to select the number of components
- Support-Vector Machines (SVMs)

Cover’s theorem

Theorem: If the classes represented in $\mathcal{T} \subset \mathbb{R}^N$ are not linearly separable, then there is a nonlinear mapping $\mu$ such that $\mu(\mathcal{T}) = \{\mu(s) \mid s \in \mathcal{T}\}$ are linearly separable.

Proof:

- Choose an enumeration $\mathcal{T} = \{s^{(1)}, s^{(2)}, \ldots, s^{(K)}\}$ where $K = |\mathcal{T}|$.
- Continuously map each sample $s$ to a different unit vector $\in \mathbb{R}^K$;
- that is, $\forall k$, $\mu(s^{(k)}) = e^{(k)}$, where $e^{(k)}_k = 1$ and $e^{(k)}_j = 0 \ \forall j \neq k$.
- For example, use Lagrange interpolating polynomials with 2-norm $\| \cdot \|$ in $\mathbb{R}^N$:

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

where $\mu = [\mu_1, \ldots, \mu_K]^T : \mathbb{R}^N \to \mathbb{R}^K$. 

© Jan. 2020 George Kesidis
Proof of Cover’s theorem (cont)

• Every partition of the samples \( \mu(T) = \{ \mu(s) \mid s \in T \} \) into two different sets (classes) \( \kappa_1 \) and \( \kappa_2 \) is separable by the hyperplane with parameters

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

• Thus, \( \forall k \in \kappa_1, \ w^T e^{(k)} = 1 > 0 \), and \( \forall k \in \kappa_2, \ w^T e^{(k)} = -1 < 0 \).

• We can build a classifier for \( C > 2 \) classes from \( C \) such linear, binary classifiers:
  – Consider partition \( \kappa_1, \kappa_2, \ldots, \kappa_C \) of \( \mu(T) \).
  – \( i^{th} \) binary classifier separates \( \kappa_i \) from \( \bigcup_{j \neq i} \kappa_j \), i.e., “one versus rest”.

• Q.E.D.

Cover’s theorem - Remarks

• Here, \( \mu(s) \) may be analogous to DNN mapping from input \( s \) to an internal layer.

• One can roughly conclude from Cover’s theorem that:

• 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 \( T \) will likely already be linearly separable.
DNN architectures

Outline:

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

Simple illustrative 2-class feed-forward neural network

Inferred Class Decision, $\hat{c}(s) = \text{argmax}_i f_i$
Some types of neurons

- Consider a neuron/unit \( n \) in layer \( \ell(n) \), \( n \in \ell(n) \), with input edge-weights \( w_{i,n} \), where neurons \( i \) are in layer prior (closer to the input) to that of \( n \), \( i \in \ell_-(n) \).

- 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

![Sigmoid activation function graph](image)
Some types of neurons (cont)

- Neurons of the sigmoid type have activation functions that include
  \[ f(z, b_n) = \tanh(zb_{n,1} + b_{n,0}) \in (-1, 1), \]  or  
  \[ f(z, b_n) = \frac{1}{1 + \exp(-zb_{n,1} - b_{n,0})} \in (0, 1), \] where \( b_{n,1} > 0 \).

- Rectified Linear activations Units (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 not necessarily bounded, whereas sigmoids are.

- “Hard threshold” neural activations involving unit-step functions \( u(x) = 1\{x \geq 0\} \),
  e.g., \( f(z, b_n) = b_{n,0}u(z - b_{n,1}) \geq 0 \), obviously are not differentiable.

- Spiking and gated neuron types are discussed later.

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 \( \ell \) is said to be fully interconnected.
Max-pooling layer - example of two partition elements $A_-, A$

- Pooling layers are intended to downsample from a large layer $\ell_-$ to a smaller one $\ell$, i.e., $|\ell_-| \gg |\ell|$.
- Each number above is a neural network activation of a max pooling layer, where
  - $|\ell_-| = 16$ (left), $|\ell| = 4$, and
  - the window of size 4 slides across the larger representation ($\ell_-$ at left) according to the stride parameter (2) to take $|\ell| = 4$ different maximum readings and form the downsampled layer $\ell$.

Convolutional layers

- Consider neurons $n \in \ell$ and suppose the neurons in layer $\ell$ and in the previous layer $\ell_-$ (or $\ell_-(n)$) are somehow ordered and enumerated.
- Let $K = \max\{|\ell_-|, |\ell|\}$.
- Layer $\ell$ is said to be **convolutional** if, $\forall n \in \ell$, its activations are:
  \[
  v_n = f \left( \sum_{i \in \ell_-(n)} v_i w_{(n-i)\mod K} \cdot b_n \right)
  \]
  where $w$ is the $K$-dimensional convolution kernel or filter.
- Two-dimensional convolutions are used in cases where the data are images.
- Compared to fully connected layers with $|\ell_-| \cdot |\ell|$ parameters, convolutional layers are “regularized” with only $K$ parameters.
- Convolutions are characteristic of linear, time-invariant transformations, which were used for decades in data processing prior to their incorporation into neural networks, and continue to be used today.
Graph-convolutional layers - an example

- Consider a layer $\ell_-$ with activations $v_i, i \in \ell_-\$
- Suppose that there is a sense of Boolean adjacency, $A_{i,k} \in \{0,1\}$ $\forall i \neq k \in \ell_-\$
- For each note $i \in \ell_-$, we can consider a neighborhood $\mathcal{N}_i(r)$ of radius $r$,
  - i.e., for all $k \in \mathcal{N}_i(r)$, there are $j_0, j_2, \ldots, j_r \in \ell_-$ such that $j_0 = i$, $j_r = k$, and $A_{j_m,j_{m+1}} = 1$ for all $m = 0, \ldots, r - 1$ (there is a path from $i$ to $k$ of length $\leq r$).
- For an example graphical layer of radius $r$, we can define the activations of the next layer $\ell$ as, $\forall n \in \ell$,
  \[
v_n = \sum_{i \in \ell_-} w_{i,n} f \left( \sum_{k \in \mathcal{N}_i(r)} b_{i,k} v_k \right),\]
  where, e.g., $f$ is a sigmoid or ReLU.
- Here, the parameters to be learned $w, b$ may be simplified so that, e.g., $\forall i, k, b_{i,k} = b_{|i-k|}$ where $|i - k| \leq r$ is the minimum path length between $i$ and $k$.

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 $\kappa \in \{1,2,\ldots,C\}$.
- The final-layer activations are, e.g.,
  \[f_\kappa(z) = \phi(\|z - b_\kappa\|^2)\]
  where $\phi$ 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 $C$-component Gaussian Mixture Model (GMM) with identity covariances and hard assignments to components.
- For nearest-prototype final layer, the class decision is the minimum $\hat{c}$ 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 \( C \) outputs \( f_\kappa \) for class \( \kappa \in \{1, 2, ..., C\} \).

- If \( f_\kappa(s) \geq 0 \) for all (DNN inputs) \( s \), then we may define, e.g.,
  \[
  \hat{p}(\kappa|s) = \frac{f_\kappa(s)}{\sum_{j=1}^{C} f_j(s)},
  \]
  else we may define, e.g.,
  \[
  \hat{p}(\kappa|s) = \frac{\exp(b f_\kappa(s))}{\sum_{j=1}^{C} \exp(b f_j(s))} \quad \text{and} \quad b > 0.
  \]

- These terms approximate the true posterior class probabilities and form the output of a softmax layer.

- For each input \( s \), a winner take all output layer gives the class decision
  \[
  \hat{c}(s) = \arg\max_{\kappa} p(\kappa|s) = \arg\max_{\kappa} f_\kappa(s).
  \]

Softmax layer - classification confidence

- Classification “confidence” can be defined as
  \[
  \frac{p_{\hat{c}(s)}(s) - \max_{\kappa \neq \hat{c}(s)} p_\kappa(s)}{p_{\hat{c}(s)}(s)} \in [0, 1].
  \]

- The class decision for \( s \) may not be accepted unless it has some “margin” \( \mu_\kappa > 0 \), i.e., unless
  \[
  \frac{p_{\hat{c}(s)}(s) - \max_{\kappa \neq \hat{c}(s)} p_\kappa(s)}{p_{\hat{c}(s)}(s)} > \mu_{\hat{c}(s)}.
  \]

- The parameters \( \mu_\kappa \) could be set using a labelled validation set \( \mathcal{H} \) which is held out of training.

• The final layer is nearest-prototype using Euclidean RBFs ("Gaussian").

• See the “explanation” near Equ. (8) of [Y. LeCun et al., 1998].

Figure 2 of Y. LeCun et al. Gradient Based Learning Applied to Document Recognition. Proc. IEEE, Nov. 1998.

Example DNN architectures - ResNet*

DNN architectures - Discussion

- The “front end” performs abstract feature extraction, e.g., convolutional 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 interpretation of class-decision confidence.

Optimization methods for training

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

- Learning objective is to choose classifier parameters $x = (w, b)$, i.e., train the classifier, to minimize the following “loss” expressions over the DNN parameters on which final-layer activations $f_\kappa$, $\kappa \in \{1, 2, ..., C\}$ and inferred class decisions $\hat{c} = \arg \max_\kappa f_\kappa$, implicitly depend.

- The training-set misclassification-rate objective,
  $$L(x) = \frac{1}{|T|} \sum_{s \in T} 1\{\hat{c}(s) \neq c(s)\},$$
  is not differentiable, and so would not lend itself to training by gradient based methods.

- A Mean Square Error (MSE) loss objective is,
  $$L(x) = \frac{1}{|T|} \sum_{s \in T} |\hat{c}(s) - c(s)|^2.$$

- Note how MSE depends on the numerical values assigned to class labels while the misclassification rate could be based on categorical class labels.

- MSE is used for regression or time-series prediction by deep learning.

Cross-entropy loss objective

- A cross-entropy loss objective is, e.g.,
  $$L(x) = -\frac{1}{|T|} \sum_{s \in T} 1\{\hat{c}(s) = c(s)\} \log \hat{p}(c(s)|s),$$
  where the DNN’s class posterior
  $$\hat{p}(\kappa|s) = \frac{f_\kappa(s)}{\sum_j f_j(s)}$$
  and softmax activations $f_j \geq 0$ are differentiable w.r.t. DNN parameters $x$.

- To simplify for deep learning, we can replace $\hat{p}(\kappa|s)$ with $f_\kappa(s)$ in the expression for $L$.

- Cross-entropy loss objectives are commonly used and optimized using gradient-based methods.

- Note that the class labels can be categorical here.
Promoting sparcity in DNN parameters

- Initially, a DNN may have excess parameters for the particular training task under consideration for it.
- Using excess parameters may result in overfitting to the training set.
- Note that the number of non-zero elements of $x_i$,
  $$\sum_i 1\{x_i \neq 0\} = \lim_{q \downarrow 0} \sum_i |x_i|^q$$
- So, one way to promote sparcity among excess parameters (i.e., zeroing them out) is to suitably penalize the optimization objective with an approximate “0-norm” penalty term,
  $$L(x) + \lambda \sum_i |x_i|^q, \text{ where } 0 < q \ll 1.$$  
- Here, reducing $q > 0$ and increasing the penalty parameter $\lambda > 0$ promotes more sparcity in the DNN parameters $x$.
- The number of non-zero elements itself is not as useful as it’s not differentiable, and so would not lend itself to training by gradient based methods.
- Note: model-order control by BIC/MDL in other learning settings.
- cf. discussion of overfitting and DNN regularization.

Back propagation for 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 $g(z_1, z_2)$ and define
  $$\partial_1 g = \frac{\partial g_1}{\partial z_1} \text{ and } \partial_2 g = \frac{\partial g_1}{\partial z_2}.$$  
- Now consider the following composed function of three variables
  $$L(x_1, x_2, x_3) = g_3(x_3, g_2(x_2, g_1(x_1)))$$
  where $g_3, g_2$ 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, $x_k$ is the parameter associated with layer $k$ and $g_k$ is the output of layer $k$, and
- the DNN output layer is 3 and layers 1,2 are further back (toward the input).
Back propagation (cont)

- Again, \( L(x_1, x_2, x_3) = g_3(x_3, g_2(x_2, g_1(x_1))) \).

- Simply by the chain rule, the gradient of \( L \) is
  \[
  \nabla L = \begin{bmatrix}
  \frac{\partial L}{\partial x_3} \\
  \frac{\partial L}{\partial x_2} \\
  \frac{\partial L}{\partial x_1}
\end{bmatrix} = \begin{bmatrix}
  \frac{\partial g_3}{\partial g_2} \cdot \frac{\partial g_2}{\partial g_1} \\
  \frac{\partial g_3}{\partial x_2} \\
  \frac{\partial g_3}{\partial x_1}
\end{bmatrix}
\]

- Note that to compute \( \frac{\partial L}{\partial x_k} \) for \( k = 1, 2 \), one needs to compute \( \frac{\partial g_2}{\partial g_3} \), 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 \( \ell \) of a DNN, the partial derivatives with respect to parameters from layers closer to the output need to be propagated back to layer \( \ell \).

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”, cf. LSTM neurons, also may not suffer from vanishing gradient problems.
Background on gradient based methods

Outline:

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


Gradients of continuously differentiable functions

- Consider a continuously differentiable function $L : \mathbb{R}^n \rightarrow \mathbb{R}$ for integer $n \geq 1$.
- Our objective is to find a local minimum $\bar{x} \in \mathbb{R}^n$ 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 $\nabla L : \mathbb{R}^n \rightarrow \mathbb{R}^n$. 
Directional derivatives

- The directional derivative of $L$ at $x$ in the direction $h$ is
  \[
  (\nabla L(x))^T h = \langle \nabla L(x), h \rangle = \lim_{\eta \to 0} \frac{L(x + \eta h) - L(x)}{\eta}.
  \]
- Here, $\eta \in \mathbb{R}$, $x, h \in \mathbb{R}^n$.
- $h$ is a descent direction 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
  \[
  \frac{L(x + \eta h) - L(x)}{\eta} - \langle \nabla L(x), h \rangle \leq \frac{1}{2} \langle \nabla L(x), h \rangle
  \Rightarrow L(x + \eta h) - L(x) \leq \frac{\eta}{2} \langle \nabla L(x), h \rangle < 0.
  \]

Optimality conditions - necessity

- $\hat{x}$ is a local minimum of $L$ if there is a $r > 0$ such that
  \[
  L(\hat{x}) \leq L(x)
  \text{ for all } x \in B(\hat{x}, r) = \{ y : \| y - \hat{x} \| < r \} \text{ (open ball centered at } \hat{x} \text{ with radius } r \}.
  \]
- **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 $H : \mathbb{R}^n \to \mathbb{R}^{n \times n}$. 
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
  \[ L(y) - L(x) = \langle \nabla L(x), y - x \rangle + \int_0^1 (1 - s)(y - x, H(x + s(y - x))(y - x))ds \]
- 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 mininizer.
  - 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 (when $n > 1$) saddle points

- local minimum: $\nabla L = 0$ and $H$ is positive definite
- local maximum: $\nabla L = 0$ and $H$ is negative definite
- saddle: $\nabla L = 0$ and $H$ is neither (dimension $n > 1$)


Gradient methods for local optimization

- To find a local minimizer, we could:
  1. try to solve $\nabla L(x) = 0$ by Newton-Raphson,
  2. then assess whether the solution is a local minimum, maximum or saddle by considering the Hessian,
  3. if not a local minimum or doesn’t converge, restart Newton-Raphson at another initial point (perhaps chosen at random).

- An advantage of gradient based methods is they do not require require higher order derivatives ($H$), or estimates of them (BFGS or DFP quasi-Newton methods).

- Disadvantages of gradient based methods is that they tend to converge slowly compared to Newton-Raphson and may converge to saddle points.
Steepest (Gradient) Descent

1. initially, $x_0 \in \mathbb{R}^n$, iteration index $k = 0$, small $\epsilon > 0$
2. if $\|\nabla L(x_k)\| < \epsilon$ 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 $\eta^*$ is a one-dimensional optimization problem.

Line search terminates at point where $-\eta \nabla L(x_k)$ is tangent to a level-set of $L$. 

© Jan. 2020 George Kesidis
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 Neural Networks

- Approaches that leverage second-order derivatives (Newton-Raphson) or their approximations (BFGS or DFP quasi-Newton methods) converge more rapidly than gradient descent,

- but these 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.

- Note that to minimize non-negative loss objectives $L \geq 0$, may terminate gradient descent when $L \leq \varepsilon$ for some small $\varepsilon > 0$. 
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**.

- This said, \( \eta \) may change dynamically, particularly \( \eta \) becomes smaller as iteration index \( k \) increases for greater "depth" of search,

- as opposed to greater "breadth" with larger \( \eta \) initially (when \( k \) is small).

- Typically, chosen learning rate \( \eta \in [0.01 - 0.99] \), e.g., \( \eta = 0.1 \).

---

**Stochastic Gradient Descent (SGD)**

- Suppose an additive loss objective to be minimized

  \[
  L(x) = \frac{1}{J} \sum_{j=1}^{J} g_j(x).
  \]

- When \( J \gg 1 \), computing \( \nabla L(x) \) at each \( x \) can be very costly.

- Instead, at step \( k \) use, e.g., search direction \( h_k = -\nabla g_{(k \mod J)}(x_k) \),

  or choose \( h_k = -\nabla g_j(x_k) \) for randomly chosen \( j \).

- Note that such \( h_k \) might not be descent direction for \( L \! \).

- Alternatively, compute the average gradient over a small random batch \( B_k \subset \{1, 2, ..., J\} \) \((|B_k| \ll J \text{ and the } B_k \text{ are i.i.d.})\), so that

  \[
  h_k = -\frac{1}{|B_k|} \sum_{j \in B_k} \nabla g_j(x_k).
  \]

- Also, at each iteration, only a randomly selected fraction of DNN parameters are updated under "random dropout".
Review of first-order autoregressive estimators

• Suppose we want to iteratively estimate the mean of the possibly nonstationary sequence $X_n$, for $n \in \{0, 1, 2, \ldots\}$, 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 $X_n$.

• An order-1 autoregressive estimator (AR-1) is

$$X_n = \alpha X_{n-1} + (1 - \alpha) X_n$$

where $0 < \alpha < 1$ is the forgetting/fading factor and $X_0 = X_0$.

AR-1 estimators (cont)

• Note that all past values of $X_i$ contribute to the current value of this autoregressive process according to weights that exponentially diminish:

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

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

$$X_n = X_{n-1} + (1 - \alpha) (X_n - X_{n-1}),$$

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 $\alpha$.

• A small $\alpha$ implies that $X_n$ 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 $X$.

• A large value of $\alpha$ means that the AR-1 process will have diminished oscillations ("low-pass" filter) but will be less responsive to changes in the distribution of the samples $X_k$. 
AR-1 estimators - example

- Suppose the initial distribution is uniform on the interval \([0, 1]\) (i.e., \(E X = 0.5\)), but for \(n \geq 20\) the distribution is uniform on the interval \([3, 4]\) (i.e., \(E X\) changes to 3.5).

- When \(\alpha = 0.2\), a sample path of the first-order AR-1 process \(X\) 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 \(\alpha = 0.8\).

\[
\begin{align*}
H_k &= \alpha H_{k-1} + (1 - \alpha) h_k \\
H_{k-1} &= \text{the search direction used for the previous set of DNN parameters, } x_{k-1}.
\end{align*}
\]

SGD with momentum

- Momentum incorporates information from prior “stochastic gradients” \(h_j\) for \(j < k\) to try to improve possibly crude approximations \(h_k\) of \(-\nabla L(x_k)\).

- 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, \(x_{k-1}\).

- Thus, \(x_k = x_{k-1} + \eta H_k\).

- To further simplify in this land of heuristics, take

\[
x_k = x_{k-1} + H_k \quad \text{with} \quad H_k = \alpha H_{k-1} + \eta h_{k-1}.
\]

- 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 minimum number parameters used to explain the training data, i.e., avoid overfitting to the training set $T$.

- Note that *a priori* no idea how many parameters suitable for very complex training datasets (large number of samples in large feature dimensions), and number of DNN parameters can be very large.

- So DNN may or may not be (initially) be overparameterized.

- Low training set accuracy may indicate too few parameters (insufficient DNN capacity to learn), while low validation set accuracy with high training set accuracy may indicate too many parameters (overfitting to the training set).

- Can heuristically reduce the number parameters, e.g., by randomly removing neurons during training, while checking performance on a validation set.

- Recall promoting neuron/edge sparcity discussion.

- *cf.* preventing overfitting by augmenting/modifying training set.

---

Overfitting illustrated*

- “Appropriate” fitting has better generalization performance on the test set by not overfitting to the two “X” class outliers.

- Held-out evaluation sets $H$ could be used to navigate between underfitting and overfitting.

---

Training dataset augmentation & batch normalization

- 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, 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.

- Also to improve generalization performance, the training dataset can be batch normalized:
  - The mean $\mu_i = \frac{1}{|\mathcal{T}|} \sum_{s \in \mathcal{T}} s_i$ and variance $\sigma_i^2 = \frac{1}{|\mathcal{T}| - 1} \sum_{s \in \mathcal{T}} (s_i - \mu_i)^2$ of all sample features indexed $i$ is computed across the training dataset $\mathcal{T}$, and
  - each training sample $s$ with features denoted $s_i$ is replaced or augmented with one whose features are $(s_i - \mu_i)/\sigma_i$ for all $i$.

- Batch normalization can also be done on internal DNN layers where all neural activations $x$ of a layer $\ell$ are adjusted by subtracting their mean and dividing by their standard deviation (as computed over the training dataset), but this complicates the neural activation functions.

Held-out validation set for hyperparameters

- Some training samples used to set “main” classifier parameters $x = (w, b)$ (by gradient based learning), 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 controlling how classifier structure is simplified (random drop-out rate during training and/or drop-out post training), and
  - parameters for representing and preprocessing data (before training).

- Also, can “early stop” training when accuracy degrades on $\mathcal{H}$ 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 $(\mathcal{T} \setminus \mathcal{H})$ is used to learn the main classifier parameters $x$. 

© Jan. 2020 George Kesidis
Spiking and Gated Neurons

Outline:

• Spiking neurons
• Dependent training samples and gated neurons
• LSTM neurons of some recurrent neural networks
• Back propagation through time

Spiking neuron types - example

• Consider a thin rectangular pulse (spike) at the origin,
  \[ \pi(t) = u(t) - u(t - \varepsilon), \]
  were \( u \) is the unit-step and positive width \( \varepsilon \ll 1 \).

• Suppose time-varying activations \( v_n(t) \) are given by the solution to the following first-order ODE,
  \[
  \frac{d\xi_n(t)}{dt} = a \cdot \left( \sum_{i \in \mathcal{T}(n)} v_i(t)w_{i,n} - \xi_n(t) \right)
  \]
  \[ v_n(t) = \sum_k \pi \left( t - \frac{k}{f(\xi_n(t), b_n)} \right) \]
  where parameter \( a \in (0, 1] \) and positive sigmoid \( f \).

• So, the activation \( v_n(t) \) at time \( t \) is a pulse train \( \pi \) with rate \( f(\xi_n(t), b_n) \).
Spiking neuron types - example (cont)

- Note that the solution to $\frac{d\xi}{dt} = a(y - \xi)$ is

$$\xi(t) = e^{-at}\xi(0) + a \int_0^t e^{-a(t-\tau)}y(\tau)d\tau$$

- So, the superposed pulse train $y$ is "smoothened-out" to determine $\xi$ and, in turn, the activation frequency $f(\xi,b)$.

- The constant-rate neural activations $v$ of the input layer directly correspond to the features of the current sample $s$, which was applied at time zero.

- In practice, the spiking activation may be numerically simulated, e.g., by Euler’s method, to solve the ODE.

---

Spiking neuron types (cont)

- One can train a DNN of such spiking neurons by porting parameters of a trained DNN with the same topology and activation functions but with the usual constant (non-spiking) signals.

- In this case, the parameters $\varepsilon, a, \xi(0)$ (which may be neuron $(n)$ dependent) may be tuned, e.g., to ensure over the training set that every pulse train’s duty cycle is smaller than its period, i.e.,

$$\varepsilon < \min_{n,t} \frac{1}{f(\xi_n(t), b_n)}.$$ 

- For distributed inference, a DNN may be partitioned into multiple elements, e.g., along edge-cuts with least average-signal magnitudes during training.

- A potential benefit of spiking DNNs is that, for high-speed inference, such distributed elements need not be very carefully synchronized.
Dependent training samples

- Classifiers may be subjected to a dependent sequence of input patterns.
- For example, a sequence of: images of a video, sounds of a speech, or words of a document.
- Classical approaches include Hidden Markov Models (HMMs).
- That is, each sample $s \in T$ may itself be a time series of dependent input patterns to the DNN, i.e.,
  
  $$s = \{s(1), s(2), \ldots, s(T_s)\} = \{s(t)\}_{t=1}^{T_s}.$$  

- The final class decision for $s$ is that which is made, e.g., when the last input pattern $s(T_s)$ is applied to the DNN.

Gated neurons

- An example memoried neuron $n$ using a simple first-order autoregressive mechanism with forgetting/fading-factor $\phi_n \in (0, 1)$ is:
  
  $$v_n(t) = \phi_nv_n(t-1) + (1 - \phi_n)f\left(\sum_{i \in \ell(n)} v_i(t)w_{i,n} + b_n\right),$$  

  where $\ell(n)$ is the layer prior to that of $n$. 
Long/Short-Term Memoried (LSTM) neurons

- For neurons $n \in \ell$, now suppose that the forgetting factor $\phi_n$ itself is dynamic (changes from sample to sample as indexed by $t$) and potentially depends on all activations $v_i(t)$ for $i \in \ell_-(n)$ (current sample $t$, previous layer $\ell_-$) and $v_k(t-1)$ for $k \in \ell, k \neq n$ (previous sample $t-1$, currently layer $\ell$).

- That is, the activations of the previous sample are stored and used (gated) when it comes time to compute the next sample.

- Consider a LSTM layer with “Minimal Gated Units” (MGUs) using positive sigmoid, e.g.,

$$f(z, b) = \frac{1}{1 + \exp(-(b_1 z + b_0))} \in (0, 1)$$

with $b > 0$,

and some other activation function $g$.

- For all neurons $n \in \ell$,

$$\phi_n(t) = f \left( \sum_{i \in \ell_-(n)} v_i(t)w_{i,n}^{(o)} + \sum_{k \in \ell, k \neq n} v_k(t-1)w_{k,n}^{(o)}, b_n^{(o)} \right)$$

$$v_n(t) = \phi_n(t)v_n(t-1) + (1 - \phi_n(t))g \left( \sum_{i \in \ell_-(n)} v_i(t)w_{i,n}^{(v)} + v_n(t-1)\phi_n(t)w_{n,n}^{(v)}, b_n^{(v)} \right)$$

- Higher order autoregression and more complex LSTM neurons are in use.

Training LSTMs

- Recall the cross-entropy loss function as

$$L(x) = -\frac{1}{|T|} \sum_{s \in T} 1\{\tilde{c}(s) = c(s)\} \log \tilde{p}(c(s)).$$

- Note that the activations for the previous sample $t-1$, $v_n(t-1)$, are needed to compute the gradient summand at time $t$, i.e., “back propagation through time”.

© Jan. 2020 George Kesidis
Concluding comments

- DNNs and the datasets they classify are extremely complex and large-scale.
- DNNs have highly heterogeneous architectures and are highly nonconvex and nonlinear.
- Class partitions in ”raw” input feature space ($\mathbb{R}^N$) are highly nonconvex.
- In practice, optimization mechanisms used, and neural and network-architectural choices made, are heuristic, trial-and-error affairs*, 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 $\mathcal{I}$, requiring actual domain expertise, may be much more time-consuming and costly than DNN training/inference.†
- An interesting history of neural networks is here:
  http://people.idsia.ch/~juergen/deep-learning-conspiracy.html