COMP4680/8650: Advanced Topics in Machine Learning

Weeks 9–10 — Deep Learning

Stephen Gould
`stephen.gould@anu.edu.au`

Australian National University

Semester 2, 2022

Optimization and Machine Learning

Generic mathematical optimization problems have the form

$$\begin{array}{rl} \text{minimize} & f_0(x) \\ \text{subject to} & f_i(x) \leq b_i, \quad i = 1, \ldots, m \end{array}$$

The solution $x^\star = (x_1^\star, \ldots, x_n^\star)$ can be:

- investment amounts in portfolio optimization
- device sizes in electronic circuits
- equipment settings in manufacturing
- **model parameters θ in machine learning**, e.g.,
  - probability distribution, $p_\theta : \mathcal{A} \to [0, 1]$
  - prediction function, $h_\theta : \mathcal{X} \to \mathcal{Y}$

Deep learning is a sub-field of machine learning focusing on *end-to-end* learnable models based on artificial neural networks.

Multilayer Perceptron (MLP)

- most basic feedforward artificial neural network, originating from work of Rosenblatt (1961) and Widrow & Hoff (1960)
- input $x \in \mathbb{R}^{p_0}$, hidden layers $z_i \in \mathbb{R}^{p_i}$, output $y \in \mathbb{R}^{p_n}$
- each layer computes its output as

$$z_i = \tilde{f}_i(z_{i-1}) = f_i(A_i z_{i-1} + b_i)$$

  where $A_i \in \mathbb{R}^{p_i \times p_{i-1}}$ and $b_i \in \mathbb{R}^{p_i}$ are parameters and f_i is a non-linear (elementwise) activation function ($x \triangleq z_0$, $y \triangleq z_n$)
- the network output is then the composition

$$\begin{aligned} y &= (\tilde{f}_n \circ \cdots \circ \tilde{f}_2 \circ \tilde{f}_1)(x) \\ &= f_n(A_n f_{n-1}(\cdots f_1(A_1 x + b_1)) + b_n) \end{aligned}$$

Activation Functions

- logistic function (sigmoid)

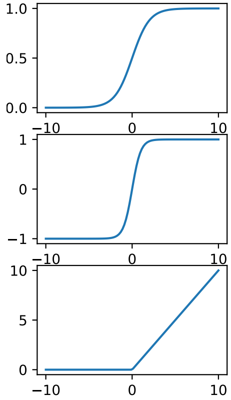
$$f(x) = \frac{1}{1+e^{-x}}$$

- hyperbolic tangent

$$f(x) = \tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$$

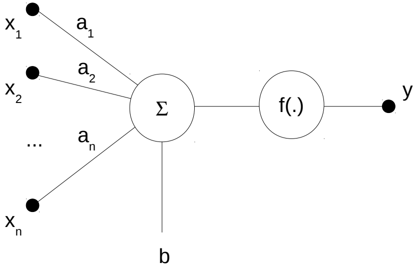
- rectified linear unit (ReLU)

$$f(x) = \max\{0, x\}$$



Single Neuron

- a single neuron computes $y = f(\sum_{i=1}^{n} a_i x_i + b) \in \mathbb{R}$



(Supervised) Learning Review

- parametrized model $y = h_\theta(x)$ and training set $\mathcal{D} = \{(x_i, y_i)\}$
- regularized loss function $\mathcal{L}$ that measures how well the model fits the training data, usually decomposes over the training data plus a prior over parameters

$$\mathcal{L}(\theta) = \sum_i \ell(h_\theta(x_i), y_i) + R(\theta)$$

- we then optimize the loss function with respect to the model parameters

$$\theta^\star = \operatorname*{argmin}_\theta \mathcal{L}$$

- simplest optimization algorithm is then gradient descent

$$\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}$$

Back-propagation

- consider a single neuron $y = f(a^T x + b)$
- if we have $\frac{\partial \mathcal{L}}{\partial y}$ we can compute $\frac{\partial \mathcal{L}}{\partial a_i}$, by the **chain rule**, as

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial a_i} &= \frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial a_i} \\ &= \frac{\partial \mathcal{L}}{\partial y} f'(a^T x + b) x_i\end{aligned}$$

- likewise we can compute $\frac{\partial \mathcal{L}}{\partial b}$ and $\frac{\partial \mathcal{L}}{\partial x_i}$

Back-propagation: Why compute $\frac{\partial \mathcal{L}}{\partial z_i}$?

- now consider a composition of neurons

$$z_i = f_i(A_i z_{i-1} + b_i), \quad i = 1, \dots, n$$

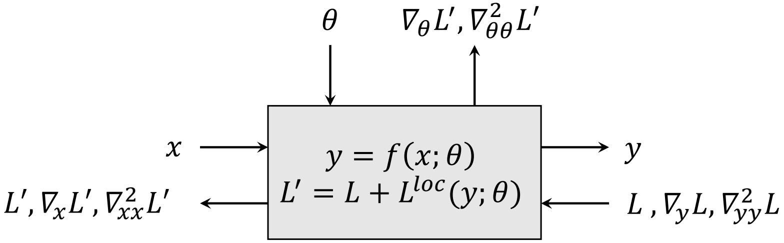
- we care about updating all parameters of the network

$$\{(A_i, b_i) \mid i = 1, \dots, n\}$$

- if we have $\nabla_{z_i} \mathcal{L}$ we can update A_i and b_i
- to update A_{i-1} and b_{i-1} we need $\nabla_{z_{i-1}} \mathcal{L}$
- so computing $\frac{\partial \mathcal{L}}{\partial z_i}$ allows us to propagate gradients backwards through the network

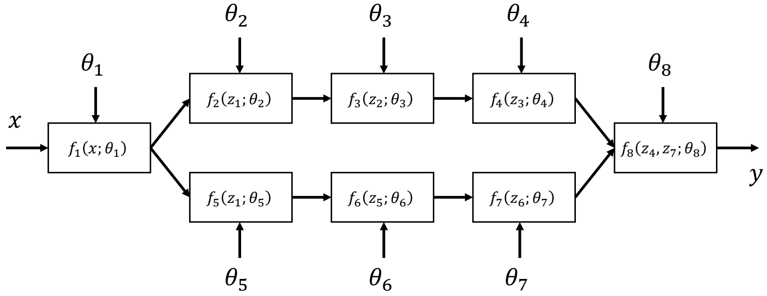
Abstract Processing Node/Layer

- a neural network is a computation graph of processing nodes
- data (or gradient) tensors pass along edges between nodes

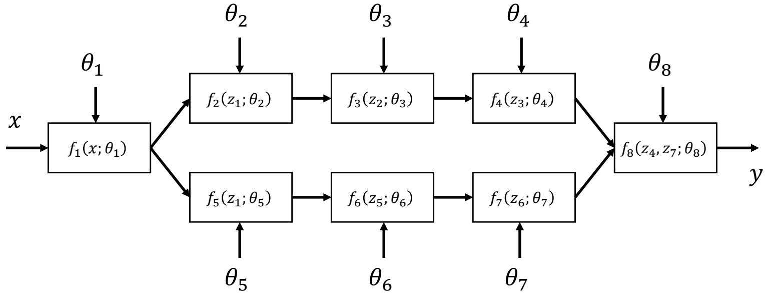


θ
$\nabla_\theta L', \nabla^2_{\theta\theta} L'$
x
$y = f(x;\theta)$
$L' = L + L^{loc}(y;\theta)$
y
$L', \nabla_x L', \nabla^2_{xx} L'$
$L\,, \nabla_y L, \nabla^2_{yy} L$

End-to-end Computation Graph



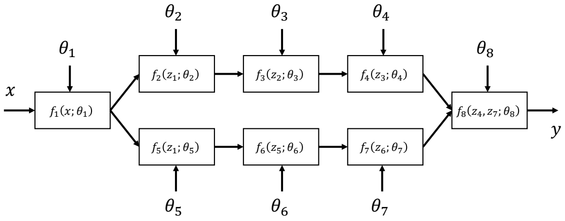
End-to-end Computation Graph



$$y = f_8\left(f_4\left(f_3\left(f_2\left(f_1(x)\right)\right)\right), f_7\left(f_6\left(f_5\left(f_1(x)\right)\right)\right)\right)$$

(omitting parameters θ_i)

End-to-end Computation Graph Gradients



Example 1.

$$\frac{\partial \mathcal{L}}{\partial \theta_7} = \frac{\partial \mathcal{L}}{\partial y} \frac{\partial y}{\partial z_7} \frac{\partial z_7}{\partial \theta_7}$$

Example 2.

$$\frac{\partial \mathcal{L}}{\partial \theta_1} = \frac{\partial \mathcal{L}}{\partial y} \left(\frac{\partial y}{\partial z_4} \frac{\partial z_4}{\partial z_3} \frac{\partial z_3}{\partial z_2} \frac{\partial z_2}{\partial z_1} + \frac{\partial y}{\partial z_7} \frac{\partial z_7}{\partial z_6} \frac{\partial z_6}{\partial z_5} \frac{\partial z_5}{\partial z_1} \right) \frac{\partial z_1}{\partial \theta_1}$$

Back-propagation: MLP Worked Example

- two-layer perceptron with sigmoid activation and squared-loss

$$\begin{aligned} z &= \sigma(Ax + b) \\ y &= \sigma(c^T z + d) \\ \mathcal{L}(A, b, c, d) &= \frac{1}{2}(y - t)^2 \end{aligned}$$

  where $\sigma(\xi) = (1 + e^{-\xi})^{-1}$ is applied elementwise

- using $\frac{\partial \mathcal{L}}{\partial y} = y - t$ and $\sigma'(\xi) = \sigma(\xi)(1 - \sigma(\xi))$ we can compute updates as ...

Back-propagation: MLP Worked Example

- layer 2 parameters:

$$\frac{\partial \mathcal{L}}{\partial d} = (y-t)y(1-y)$$
$$\nabla_c \mathcal{L} = (y-t)y(1-y)z$$

- layer 2 input / layer 1 output:

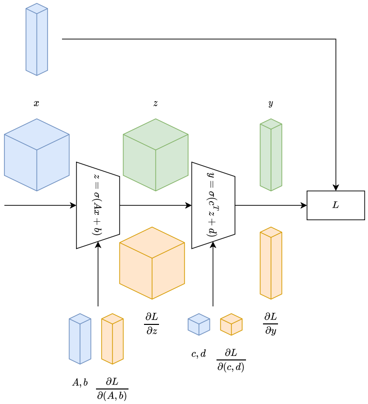
$$\nabla_z \mathcal{L} = (y-t)y(1-y)c$$

- layer 1 parameters:

$$\nabla_b \mathcal{L} = \nabla_z \mathcal{L} \circ z \circ (1-z)$$
$$\nabla_A \mathcal{L} = (y-t)y(1-y)\left(c \circ z \circ (1-z)\right) x^T$$

- note that we have reused calculations from the forward pass (memory-vs-compute trade-off)

Memory Buffers



x
z
y
$z = \sigma(Ax + b)$
$y = \sigma(c^T z + d)$
L
$\frac{\partial L}{\partial z}$
$\frac{\partial L}{\partial y}$
c, d
$\frac{\partial L}{\partial (c,d)}$
A, b
$\frac{\partial L}{\partial (A,b)}$

- in-place operations may save memory in the forward pass
- re-using buffers may save memory in the backward pass
- at *test time* buffering data is not needed

Automatic Differentiation

- automatic differentiation (AD) is an algorithmic procedure that produces code for computing the derivative of a function
- assumes numeric computations are composed of a finite set of elementary operations that we know how to differentiate
  - arithmetic, exp, log, trigonometric
- two flavours
  - (forward mode) propagate calculations of first-order approximation $x + \Delta x$ forward through the computations
  - (reverse mode) builds program to compute gradient based on the chain rule re-using computation where applicable
- different deep learning frameworks use slightly different approaches (explicit graph construction versus implicit operator tracking)
- python package **autograd** (for reverse mode AD)

Autograd Example

```
import autograd.numpy as np
from autograd import grad

# define a function
def tanh(x):
  y = np.exp(2.0 * x)
  return (y - 1.0) / (y + 1.0)

# create function to compute gradient
dtanh = grad(tanh)

# evaluate function and gradient at 1.0
print(tanh(1.0))
print(dtanh(1.0))
print((tanh(1.01) - tanh(0.99)) / 0.02)
```

Universal Approximation Theorem
(Cybenko 1989, Hornik 1991)

Theorem (roughly):

- let $f : \mathbb{R} \to \mathbb{R}$ be a (well-behaved) activation function
- let $g : \mathbb{R}^n \to \mathbb{R}$ be any continuous function
- then there exists parameters $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, and $c \in \mathbb{R}^m$ such that the function

  $$\hat{g}(x) = c^T f(Ax + b)$$

  approximates $g(x)$ everywhere
- that is, $|\hat{g}(x) - g(x)| \leq \epsilon$ for all x

Historical Notes

- neural networks were very popular from 1960s to 1990s
- non-convexity and problems with diminishing and exploding gradients made them difficult to train
- they also often had poor generalization to unseen data
- lack of theory compared to other machine learning approaches
- . . . led to the AI winter

Historical Notes

- neural networks were very popular from 1960s to 1990s
- non-convexity and problems with diminishing and exploding gradients made them difficult to train
- they also often had poor generalization to unseen data
- lack of theory compared to other machine learning approaches
- . . . led to the AI winter

- some persisted (Hinton, Lecun, Bengio, Schmidhuber, . . .)
- Krizhevsky et al. 2012 started a resurgence in NN research
- we're now in the AI spring . . .
- better training algorithms and software libraries
  - pre-training
  - stochastic gradient descent on mini-batches
  - drop-out (and other regularization incl. data augmentation)
  - batch normalization
- more labelled training data
- faster and bigger hardware (e.g., GPUs)
- still lack of theory but large community sharing best practice

Stochastic Gradient Descent (SGD)

- (batch) gradient descent on training set $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^m$ is

$$\begin{aligned}\theta &\leftarrow \theta - \eta \nabla_\theta \mathcal{L} \\ &= \theta - \eta \sum_{i=1}^{m} \nabla_\theta \ell(f(x_i; \theta), y_i)\end{aligned}$$

Stochastic Gradient Descent (SGD)

- (batch) gradient descent on training set $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^m$ is

$$\begin{aligned}\theta &\leftarrow \theta - \eta \nabla_\theta \mathcal{L} \\ &= \theta - \eta \sum_{i=1}^{m} \nabla_\theta \ell(f(x_i;\theta), y_i)\end{aligned}$$

- to save computation stochastic gradient descent approximates the gradient on mini-batches $\mathcal{I} \subseteq \{1, \ldots, m\}$

$$\widehat{\nabla_\theta \mathcal{L}} = \sum_{i \in \mathcal{I}} \nabla_\theta \ell(f(x_i;\theta), y_i)$$

Stochastic Gradient Descent (SGD)

- (batch) gradient descent on training set $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^m$ is

$$\begin{aligned}\theta &\leftarrow \theta - \eta \nabla_\theta \mathcal{L} \\ &= \theta - \eta \sum_{i=1}^{m} \nabla_\theta \ell(f(x_i; \theta), y_i)\end{aligned}$$

- to save computation stochastic gradient descent approximates the gradient on mini-batches $\mathcal{I} \subseteq \{1, \ldots, m\}$

$$\widehat{\nabla_\theta \mathcal{L}} = \sum_{i \in \mathcal{I}} \nabla_\theta \ell(f(x_i; \theta), y_i)$$

- can be accumulated over multiple forward passes

Stochastic Gradient Descent (SGD)

- (batch) gradient descent on training set $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^m$ is

$$\begin{aligned}\theta &\leftarrow \theta - \eta \nabla_\theta \mathcal{L} \\ &= \theta - \eta \sum_{i=1}^{m} \nabla_\theta \ell(f(x_i; \theta), y_i)\end{aligned}$$

- to save computation stochastic gradient descent approximates the gradient on mini-batches $\mathcal{I} \subseteq \{1, \ldots, m\}$

$$\widehat{\nabla_\theta \mathcal{L}} = \sum_{i \in \mathcal{I}} \nabla_\theta \ell(f(x_i; \theta), y_i)$$

- can be accumulated over multiple forward passes
- under mild assumptions $E\left[\widehat{\nabla_\theta \mathcal{L}}\right] = \nabla_\theta \mathcal{L}$

Stochastic Gradient Descent (SGD)

- (batch) gradient descent on training set $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^m$ is

$$\begin{aligned}\theta &\leftarrow \theta - \eta \nabla_\theta \mathcal{L} \\ &= \theta - \eta \sum_{i=1}^{m} \nabla_\theta \ell(f(x_i; \theta), y_i)\end{aligned}$$

- to save computation stochastic gradient descent approximates the gradient on mini-batches $\mathcal{I} \subseteq \{1, \ldots, m\}$

$$\widehat{\nabla_\theta \mathcal{L}} = \sum_{i \in \mathcal{I}} \nabla_\theta \ell(f(x_i; \theta), y_i)$$

- can be accumulated over multiple forward passes
- under mild assumptions $E\left[\widehat{\nabla_\theta \mathcal{L}}\right] = \nabla_\theta \mathcal{L}$
- in practice we permute $[m]$ and iterate through adjacent fixed-length intervals; once through the data is called an epoch

AdaGrad (Duchi et al., 2011)

- view gradient descent as a first-order proximal method

$$\begin{aligned}\theta^{(k+1)} &= \operatorname*{argmin}_{\theta} \left\{ \langle \nabla \mathcal{L}(\theta^{(k)}), \theta \rangle + \frac{1}{2\eta_k} \|\theta - \theta^{(k)}\|_2^2 \right\} \\ &= \theta^{(k)} - \eta_k \nabla \mathcal{L}(\theta^{(k)})\end{aligned}$$

AdaGrad (Duchi et al., 2011)

- view gradient descent as a first-order proximal method

$$
\begin{aligned}
\theta^{(k+1)} &= \operatorname*{argmin}_{\theta} \left\{ \langle \nabla \mathcal{L}(\theta^{(k)}), \theta \rangle + \frac{1}{2\eta_k} \|\theta - \theta^{(k)}\|_2^2 \right\} \\
&= \theta^{(k)} - \eta_k \nabla \mathcal{L}(\theta^{(k)})
\end{aligned}
$$

- adapt the step size (geometry) for different features to increase influence of rare but informative features

$$
\begin{aligned}
\theta^{(k+1)} &= \operatorname*{argmin}_{\theta} \left\{ \langle \nabla \mathcal{L}(\theta^{(k)}), \theta \rangle + \frac{1}{2\eta_k} \|\theta - \theta^{(k)}\|_B^2 \right\} \\
&= \theta^{(k)} - \eta_k B^{-1} \nabla \mathcal{L}(\theta^{(k)})
\end{aligned}
$$

AdaGrad (Duchi et al., 2011)

- view gradient descent as a first-order proximal method

$$\begin{aligned}\theta^{(k+1)} &= \operatorname*{argmin}_{\theta} \left\{ \langle \nabla \mathcal{L}(\theta^{(k)}), \theta \rangle + \frac{1}{2\eta_k} \|\theta - \theta^{(k)}\|_2^2 \right\} \\ &= \theta^{(k)} - \eta_k \nabla \mathcal{L}(\theta^{(k)})\end{aligned}$$

- adapt the step size (geometry) for different features to increase influence of rare but informative features

$$\begin{aligned}\theta^{(k+1)} &= \operatorname*{argmin}_{\theta} \left\{ \langle \nabla \mathcal{L}(\theta^{(k)}), \theta \rangle + \frac{1}{2\eta_k} \|\theta - \theta^{(k)}\|_B^2 \right\} \\ &= \theta^{(k)} - \eta_k B^{-1} \nabla \mathcal{L}(\theta^{(k)})\end{aligned}$$

- use previous gradients to estimate B as diagonal matrix

$$G^{(k)} = \left(\sum_{i=1}^{k} \textbf{diag} \left(\nabla \mathcal{L}(\theta^{(i)}) \right)^2 + \epsilon I \right)^{1/2}$$

AdaGrad (Duchi et al., 2011)

- view gradient descent as a first-order proximal method

$$\begin{aligned}\theta^{(k+1)} &= \underset{\theta}{\operatorname{argmin}} \left\{ \langle \nabla \mathcal{L}(\theta^{(k)}), \theta \rangle + \frac{1}{2\eta_k} \|\theta - \theta^{(k)}\|_2^2 \right\} \\ &= \theta^{(k)} - \eta_k \nabla \mathcal{L}(\theta^{(k)})\end{aligned}$$

- adapt the step size (geometry) for different features to increase influence of rare but informative features

$$\begin{aligned}\theta^{(k+1)} &= \underset{\theta}{\operatorname{argmin}} \left\{ \langle \nabla \mathcal{L}(\theta^{(k)}), \theta \rangle + \frac{1}{2\eta_k} \|\theta - \theta^{(k)}\|_B^2 \right\} \\ &= \theta^{(k)} - \eta_k B^{-1} \nabla \mathcal{L}(\theta^{(k)})\end{aligned}$$

- use previous gradients to estimate B as diagonal matrix

$$G^{(k)} = \left(\sum_{i=1}^{k} \textbf{diag} \left(\nabla \mathcal{L}(\theta^{(i)}) \right)^2 + \epsilon I \right)^{1/2}$$

- weakness: $G^{(k)}$ keeps growing so learning rate becomes infinitesimally small

Adam

(Kingma and Ba, 2014)

- maintains exponentially decaying averages of past gradients and gradients-squared

$$\begin{aligned} m^{(k)} &= \beta_1 m^{(k-1)} + (1-\beta_1)\nabla\mathcal{L}(\theta^{(k)}) \\ v^{(k)} &= \beta_2 v^{(k-1)} + (1-\beta_2)\nabla\mathcal{L}(\theta^{(k)})^2 \end{aligned}$$

Adam
(Kingma and Ba, 2014)

- maintains exponentially decaying averages of past gradients and gradients-squared

$$\begin{aligned} m^{(k)} &= \beta_1 m^{(k-1)} + (1-\beta_1)\nabla\mathcal{L}(\theta^{(k)}) \\ v^{(k)} &= \beta_2 v^{(k-1)} + (1-\beta_2)\nabla\mathcal{L}(\theta^{(k)})^2 \end{aligned}$$

- compensate for bias (c.f. unbiased variance calculations)

$$\hat{m}^{(k)} = \frac{1}{1-\beta_1^k} m^{(k)} \text{ and } \hat{v}^{(k)} = \frac{1}{1-\beta_2^k} v^{(k)}$$

Adam
(Kingma and Ba, 2014)

- maintains exponentially decaying averages of past gradients and gradients-squared

$$\begin{aligned} m^{(k)} &= \beta_1 m^{(k-1)} + (1-\beta_1)\nabla\mathcal{L}(\theta^{(k)}) \\ v^{(k)} &= \beta_2 v^{(k-1)} + (1-\beta_2)\nabla\mathcal{L}(\theta^{(k)})^2 \end{aligned}$$

- compensate for bias (c.f. unbiased variance calculations)

$$\hat{m}^{(k)} = \frac{1}{1-\beta_1^k} m^{(k)} \text{ and } \hat{v}^{(k)} = \frac{1}{1-\beta_2^k} v^{(k)}$$

- apply update rule

$$\theta^{(k+1)} = \theta^{(k)} - \eta\,\mathbf{diag}\left(\hat{v}^{(k)} + \epsilon\right)^{-1/2} \hat{m}^{(k)}$$

AdamW

(Loshchilov and Hutter, 2019)

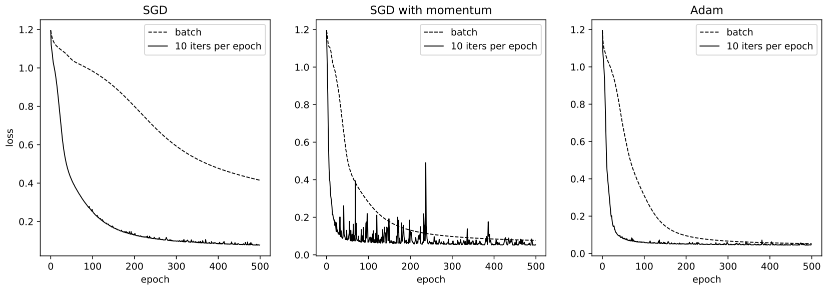
- many other variants of update rules
- for example, Adam with weight decay

$$\theta^{(k+1)} = \theta^{(k)} - \eta\, \textbf{diag}\left(\hat{v}^{(k)} + \epsilon\right)^{-1/2} \left(\hat{m}^{(k)} + w\theta^{(k)}\right)$$

- empirically works better than applying ℓ_2 regularization on $m^{(k)}$

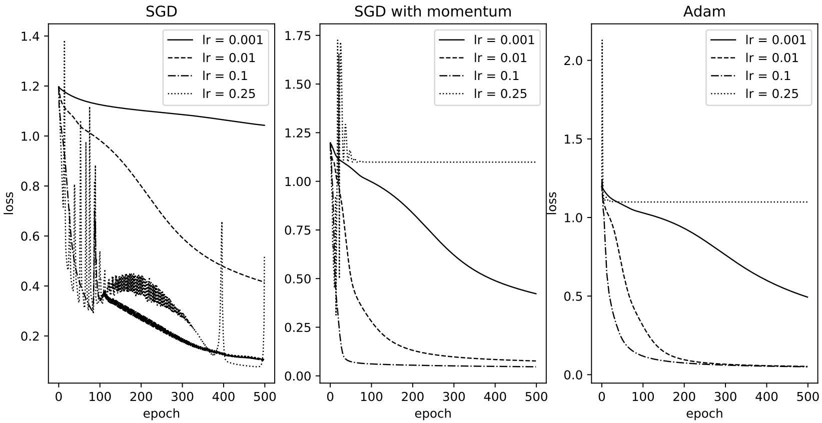
Learning Curves

- Iris dataset (3 classes, 4 features, 150 examples)
- multi-layer perceptron with:
  - 4 inputs, 8 hidden nodes (ReLU), 3 outputs (softmax)
- trained with cross-entropy loss (on all 150 examples)
- training loss plotted for different SGD variants



Learning Curves 2

- same setup as previous
- training loss plotted for different learning rates



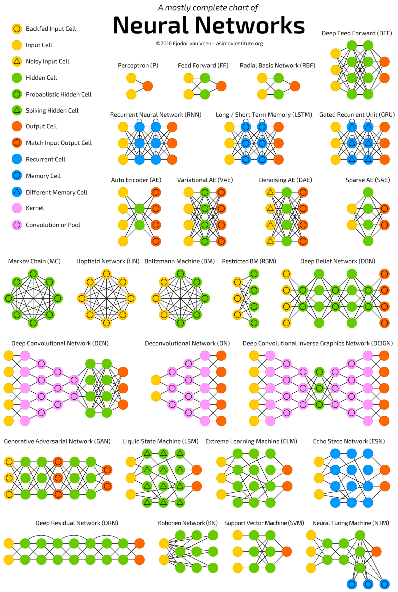
Training Tricks

Researchers have experimented with a number of tricks to help improve training/generalizability of deep models. Some of the tricks that seem to help include:

- careful parameter initialization (e.g., Xavier initialization)
- pre-training on a large annotated dataset (e.g., ImageNet)
- diverse mini-batches (e.g., randomize at start of epoch)
- data augmentation
- drop-out (not so popular anymore)
- batch normalization between layers
- gradient clipping
- intermediate supervision
- iterative refinement (skip/residual/short-cut connections)
- contrastive learning and multi-modal embedding

Model Zoo

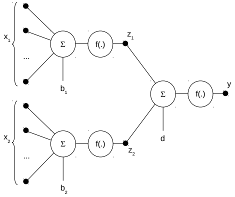
(http://www.asimovinstitute.org/neural-network-zoo/)


A mostly complete chart of
Neural Networks
©2016 Fjodor van Veen - asimovinstitute.org
Backfed Input Cell
Input Cell
Noisy Input Cell
Hidden Cell
Probablistic Hidden Cell
Spiking Hidden Cell
Output Cell
Match Input Output Cell
Recurrent Cell
Memory Cell
Different Memory Cell
Kernel
Convolution or Pool
Perceptron (P)
Feed Forward (FF)
Radial Basis Network (RBF)
Deep Feed Forward (DFF)
Recurrent Neural Network (RNN)
Long / Short Term Memory (LSTM)
Gated Recurrent Unit (GRU)
Auto Encoder (AE)
Variational AE (VAE)
Denoising AE (DAE)
Sparse AE (SAE)
Markov Chain (MC)
Hopfield Network (HN)
Boltzmann Machine (BM)
Restricted BM (RBM)
Deep Belief Network (DBN)
Deep Convolutional Network (DCN)
Deconvolutional Network (DN)
Deep Convolutional Inverse Graphics Network (DCIGN)
Generative Adversarial Network (GAN)
Liquid State Machine (LSM)
Extreme Learning Machine (ELM)
Echo State Network (ESN)
Deep Residual Network (DRN)
Kohonen Network (KN)
Support Vector Machine (SVM)
Neural Turing Machine (NTM)

Parameter Sharing

- consider a MLP acting on input $x = (x_1, x_2) \in \mathbb{R}^{2n}$

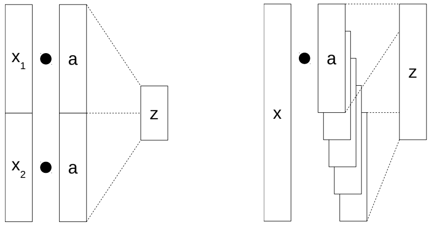
$$z = f\left(\begin{bmatrix} a_1^T & 0 \\ 0 & a_2^T \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}\right) \in \mathbb{R}^2$$
$$y = f(c^T z + d) \in \mathbb{R}$$



- now suppose we want to share weights $a_1 = a_2 = a \in \mathbb{R}^n$
- then we need to combine gradients

$$\nabla_a y = \frac{\partial y}{\partial z_1} \nabla_a z_1 + \frac{\partial y}{\partial z_2} \nabla_a z_2$$

Parameter Sharing — Convolutions



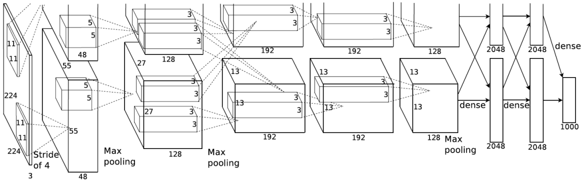
- parameters vector $a \in \mathbb{R}^h$ and input $x \in \mathbb{R}^n$
- weighted sum of inputs becomes convolution with a shared filter kernel

$$z = f(a * x + b)$$

where $z_i = f(a^T x_{i+h:i} + b)$ for $i = 1, \ldots, n - h$

Convolutional Neural Networks (CNNs)

- a CNN takes as input an image ($3 \times W \times H$ tensor) and performs successive layers of:
  - convolution, $(x * a)_{st} = \sum_{i,j} x_{s-i,t-j} \cdot a_{ij} + b$
  - non-linear transform, e.g., ReLU
  - pooling/downsampling, e.g., $z_{st} = \max\{x_{ij} \mid i, j \in \mathcal{N}_{st}\}$
- the final layers are typically "fully-connected" (i.e., an MLP)



- size of a is often called the "receptive field"
- variants: change stride/padding, dilated (atrous), decomposed, etc.

Recurrent Neural Networks

- we can feed the (partial) output of a network back into itself and time-step inference to give a recurrent network,

$$(y_t, h_t) = f(x_t, h_{t-1})$$

- useful for sequence modelling

Recurrent Neural Networks

- we can feed the (partial) output of a network back into itself and time-step inference to give a recurrent network,

$$(y_t, h_t) = f(x_t, h_{t-1})$$

- useful for sequence modelling
- most popular variant is the *long short-term memory* (LSTM) network

$$\begin{aligned}
f_t &= \sigma(W_f x_t + U_f h_{t-1} + b_f) && \text{forget activations} \\
i_t &= \sigma(W_i x_t + U_i h_{t-1} + b_i) && \text{input activations} \\
o_t &= \sigma(W_o x_t + U_o h_{t-1} + b_o) && \text{output activations} \\
c_t &= f_t \circ c_{t-1} + i_t \circ \sigma(W_c x_t + U_c h_{t-1} + b_c) && \text{cell memory} \\
h_t &= o_t \circ \sigma(c_t) && \text{cell output}
\end{aligned}$$

Back-propagation Through Time

- We can train recurrent networks by "unrolling" the network to a given time horizon and back-propagating through time.

$$
\begin{aligned}
(y_t, h_t) &= f(x_t, h_{t-1}; \theta) \\
(y_{t-1}, h_{t-1}) &= f(x_{t-1}, h_{t-2}; \theta) \\
&\vdots \\
(y_1, h_1) &= f(x_1, h_0; \theta)
\end{aligned}
$$

Back-propagation Through Time

- We can train recurrent networks by "unrolling" the network to a given time horizon and back-propagating through time.

$$
\begin{aligned}
(y_t, h_t) &= f(x_t, h_{t-1}; \theta) \\
(y_{t-1}, h_{t-1}) &= f(x_{t-1}, h_{t-2}; \theta) \\
&\vdots \\
(y_1, h_1) &= f(x_1, h_0; \theta)
\end{aligned}
$$

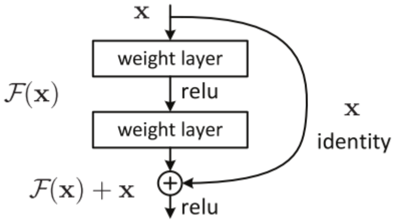
- Then (assuming the loss is applied to the last output, y_t)

$$
\frac{\partial \mathcal{L}}{\partial \theta} = \frac{\partial \mathcal{L}}{\partial y_t}\frac{\partial y_t}{\partial \theta} + \frac{\partial \mathcal{L}}{\partial y_t}\frac{\partial y_t}{\partial h_{t-1}}\frac{\partial h_{t-1}}{\partial \theta} + \cdots
$$

$$
\cdots + \frac{\partial \mathcal{L}}{\partial y_t}\frac{\partial y_t}{\partial h_{t-1}}\left(\prod_{\tau=1}^{t-1}\frac{\partial h_{\tau+1}}{\partial h_\tau}\right)\frac{\partial h_1}{\partial \theta}
$$

Short-cut Connections

- one way to address the issue of diminishing gradients is via short-cut connections
- the ResNet block is a classic example

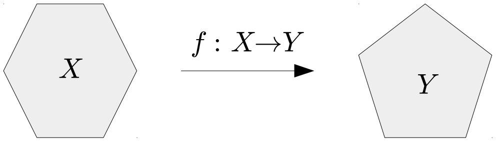


- the gradient through the ResNet block is

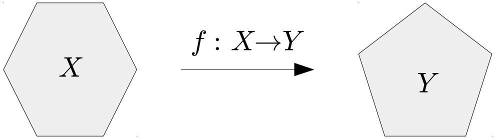
$$\frac{\partial y}{\partial x} = \mathcal{F}'(x) + 1$$

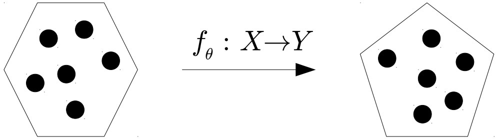
- ResNet also models “residual signals”, which may help lead to it’s empirically observed improved performance

Machine Learning: An Abstract View

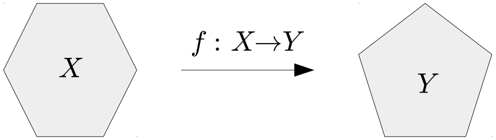


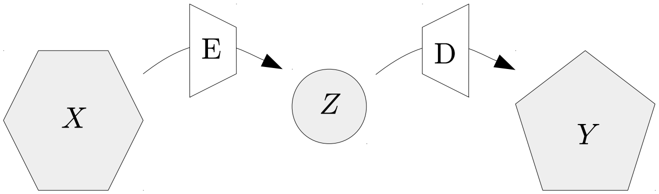
Machine Learning: An Abstract View





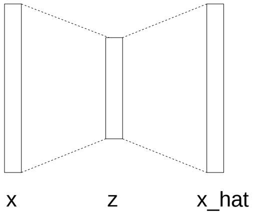
Machine Learning: Encoder-decoder Models





Auto-encoders

- attempts to learn a low-dimensional (and sometimes sparse) representation of data
- does not need explicit supervision



- encoder: $f : \mathbb{R}^n \to \mathbb{R}^m$ with $m \ll n$
- decoder: $g : \mathbb{R}^m \to \mathbb{R}^n$
- learn such that $g(f(x)) \approx x$
- then $z = f(x)$ is a good latent representation of x

Encoder-decoder Models

- we can generalize auto-encoders where the output does not need to be a reconstruction of the input
- still learned end-to-end so that the “latent” representation captures some higher level meaning
- this gives rise to many interesting applications, e.g.,
  - use a CNN encoder-decoder for semantic segmentation or style-transfer
  - use an RNN encoder-decoder for language translation (sometimes called sequence-to-sequence models)
  - use a mixed CNN-RNN encoder-decoder for image captioning
  - learn joint image-language embedding (e.g., CLIP) then generate images from natural language description (DALL-E, Imagen, Stable Diffusion)

Probabilistic Auto-encoders

- adds probability distributions over the data x and latent representation z, $p(x \mid z)$ and $p(z)$, respectively
- some things we might want to are:
  - jointly sample data and representation from $p_\theta(x, z)$
  - compute the marginal $p_\theta(x) = \int p_\theta(x \mid z) p_\theta(z) \mathrm{d}z$
  - sample representation from $p_\theta(z \mid x) = \frac{p_\theta(x|z)p_\theta(z)}{p_\theta(x)}$
  - infer (learn) parameters θ
- the first is easy, the last three are difficult (in general)
- solved by learning a distribution $q_\phi(z \mid x)$

Variational Auto-encoders (VAEs)

(Kingma and Welling, 2013; Rezende et al., 2014)

- admits efficient sampling from a learned distribution, $p_\theta(x \mid z)$,

$$z \sim \mathcal{N}(0, I)$$
$$x \sim \mathcal{N}(\mu(z), \Sigma(z))$$

  where functions $\mu(\cdot)$ and $\Sigma(\cdot)$ are learned (decoder models)
- introduce variational lower bound for jointly learning ϕ and θ

$$\log p_\theta(x) \geq \mathbf{E}_{q_\phi(z|x)}\left[\log \frac{p_\theta(x, z)}{q_\phi(z \mid x)}\right]$$

- the gradients cannot propagate through random variables[1] so we need to reparametrize x as follows

$$x \sim \mathcal{N}(\mu, \Sigma) \text{ becomes } x = \mu + L\epsilon$$

  where $\epsilon \sim N(0, I)$ and $\Sigma = LL^T$

[1]sweeping some technicalities under the carpet

Generative Adversarial Models (GANs)

(Goodfellow et al., 2014)

- alternative method for learning to generate samples based on ideas of training competing models to reach an equilibrium
- does not explicitly represent the data generating distribution
- the **generator** G_θ produces novel "realistic" samples that fool the discriminator
- the **discriminator** D_ϕ classifies samples as "real" or "fake"
- mathematically we can write the objective

$$\min_{\theta} \max_{\phi} \left[E_{x \sim p_{\text{data}}} \log D_\phi(x) + E_{z \sim p_Z} \log(1 - D_\phi(G_\theta(z)))\right]$$

- due to optimization issues, we typically replace generative loss with $-\log D_\phi(G_\theta(z))$

Attention and Transformers

(Polosukhin et al., 2017)

- recurrent models on sequences tend to have difficulties learning due to vanishing/exploding gradients (LSTMs alleviate this to some extent)
- solution is to replace recurrent network with **attention** mechanism

$$f(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{n}}\right) V$$

  where $Q \in \mathbb{R}^{m \times n}$, $K \in \mathbb{R}^{p \times n}$ and $V \in \mathbb{R}^{p \times q}$ are the "query", "key" and "value" matrices, respectively.
- add positional encoding (multi-frequency sine waves) for longer sequences
- (add bells and whistles for full transformer model)

Diffusion Models

(Ho, Jain and Abbeel, 2020)

- **forward process:**

$$x_0 \sim q(x) \to \cdots \to x_{t-1} \xrightarrow{q(x_t|x_{t-1})} x_t \to \cdots \to x_T$$

  where $q(x_t \mid x_{t-1}) = \mathcal{N}(\sqrt{1-\beta_{t-1}}x_{t-1}, \beta_{t-1}I)$

- **reverse process:**

$$x_t \sim \mathcal{N}(0, I) \to \cdots \to x_t \xrightarrow{q(x_{t-1}|x_t)} x_{t-1} \to \cdots \to x_0$$

  but where $q(x_{t-1} \mid x_t)$ is, in general, unknown

- uses variational l.b. to estimate $q(x_{t-1} \mid x_t)$ as $p_\theta(x_{t-1} \mid x_t)$
- here p_θ is a learned denoising model
- can be used for image generation (or guided image generation with a few modifications to p_θ)

Reinforcement Learning

- at each timestep an agent observes the state of the environment s_t and takes an action a_t; the environment provides a reward r_{t+1} and updates its state s_{t+1}
- r_{t+1} and s_{t+1} may be governed by stochastic processes
- reinforcement learning results in a policy π that aims to maximize the sum of discounted future rewards

$$R = \sum_{t=0}^{\infty} \gamma^t r_t$$

- we often learn a Q-function from experience

$$Q^\pi(s, a) = \mathbf{E}\left[R \mid s, a, \pi\right]$$

  which gives the expected reward R for taking action a in state s and then following policy π
- we obtain the optimal action as $a_t \in \operatorname{argmax} Q^{\pi^\star}(s_t, \cdot)$

Deep Reinforcement Learning

(Mni et al., Nature, 2015)

deep Q-learning (DQN):

- approximate $Q^{\pi^\star}$ by a neural network f_θ
- stochastic gradient update of parameters using loss

$$\mathcal{L}(\theta) = \left(\overbrace{r + \gamma \max_{a'} f_{\tilde{\theta}}(s', a')}^{\text{one action on } \tilde{Q}} - \underbrace{f_\theta(s, a)}_{\text{learned } Q} \right)^2$$

where $\tilde{\theta}$ are the fixed previous model parameters and a, s, s' and r are sampled action, state, next state, and final reward from experience, respectively

policy gradient: directly learn $a_t = f(x_t; \theta)$

- e.g., $a_t \sim p_\theta(\text{“win”} \mid a_t, s_t) \propto p_\theta(a_t \mid s_t, \text{“win”})$
- train by rolling out sampled games and weighting each action by the final total reward