Deep Learning
Pietro Michiardi

Lecture 01: Deep Neural Networks

I. Fully connected Nets

Goal: approximate a function $f^*(x)$ from data with $f(x, W)$.

Structure of the model: input, hidden, output layers; depth, width; cost function.

$X \in \mathbb{R}^{N \times D}$, $Y \in \mathbb{R}^{N \times C}$, $W^{(i)} \in \mathbb{R}^{|h^{(i-1)}| \times |h^{(i)}|}$.

Cost Function: Deep Nets are non-linear ⇒ Loss functions are not convex.

Entropy is a measure of uncertainty of a random variable. Given X with $p(x) = \Pr(X = x)$ we have:

$$H(X) = -\sum p(x) \log p(x) = -\mathbb{E}_{X \sim p(x)} \log p(x) = \mathbb{E}_{X \sim p(x)} \frac{1}{\log p(x)}$$

Kullback-Leibler divergence: divergence between two distributions:

$$KL[p(x) \parallel q(x)] = \sum p(x) \log \frac{p(x)}{q(x)} = \mathbb{E}_{X \sim p(x)} \log \frac{p(x)}{q(x)}$$

Cross-Entropy: $H\big(p(x), q(x)\big) = \mathbb{E}_{X \sim p(x)} \log \frac{1}{q(x)}$

Maximum Likelihood and Cross-Entropy: $W_{ML} = \arg\max_W q(X|W) = \arg\min_W H\big(p(x), q(x|W)\big)$

Output Units:

- Linear Units: $\hat{y} = f^{(l-1)}(h)^T W^l$
- Sigmoid Units: $\hat{y} = \sigma\big(f^{(l-1)}(h)^T W^l\big)$ used to squash the output in [0,1]
- Softmax Units: $\hat{y}_i = \text{softmax}\big(f^{(l-1)}(h)^T W^l\big) = \frac{\exp z_i}{\sum \exp z_i}$ used in multi-class classification

Hidden Units: we apply an element-wise non-linear function to the layer input.

ReLu: $g(z) = \max(0, z)$

Extension of ReLu: $g(z, \alpha) = \max(0, z) + \alpha \min(0, z)$

Regularization: modification of algo to reduce test error but not train error (to prevent overfitting)

Norm: measure of the "size" of vector: $\|x\|_p = (\sum |x_i|^p)^{\frac{1}{p}}$

We usually use squared L2 norm instead of L2 (easier to derive): $\frac{\partial \|x\|_2^2}{\partial x_1} = \frac{\partial}{\partial x_1}(x_1^2 + x_2^2) = 2x_1$.

Regularized loss: $\tilde{\mathcal{L}}(W; X, y) = \mathcal{L}(W; X, y) + \alpha\Omega(W) = \mathcal{L}(W; X, y) + \frac{\alpha}{2} W^T W$

So $\nabla_W \tilde{\mathcal{L}}(W; X, y) = \nabla_W \mathcal{L}(W; X, y) + \alpha W$.

Example: SGD with regularized updates: $W \leftarrow (1 - \lambda\alpha)W - \lambda \nabla_W \mathcal{L}(W; X, y)$

Regularization by Noise Injection (in input samples, in weights, in labels).

Regularization by Early Stopping.

Regularization by Dropout: cancel some units: $f^{(l)}(h) = g\left(\left(h^T W^{(l)}\right) \odot m^{(l)}\right)$ where m is the dropout mask.

Regularization by DropConnect: cancel some connections: $f^{(l)}(h) = g\left(h^T\left(W^{(l)} \odot M^{(l)}\right)\right)$

II. Loss Landscapes

The ability to train depend on the initialization, the depth (use skip connection and more units per layer).

Hessian is positive semi-defined → convex or semi-convex

III. Stochastic Optimization

Challenges: differentiability, non-convex surfaces, saddle points (gradient is zero), exploding/vanishing gradient (if λ_i are not around 1 in $W = Vdiag(\lambda)V^{-1}$)

Basic algos

- Gradient Descent: all samples $W_{t+1} = W_t - \lambda_t \nabla \mathcal{J}(W_t, X, y)$
- Stochastic Gradient Descent: one sample $W_{t+1} = W_t - \lambda_t \nabla \mathcal{J}(W_t, x_i, y_i)$
- Mini-batch Stochastic Gradient Descent: a random batch $W_{t+1} = W_t - \lambda_t \frac{1}{\mathrm{m}} \sum_i \nabla \mathcal{J}(W_t, x_i, y_i)$

To guarantee convergence $\sum_i \lambda_t = \infty$ and $\sum_i \lambda_t^2 < \infty$.
Momentum to store the direction and speed at which W move: $v_{t+1} = \alpha v_t + \lambda g_t$ and $W_{t+1} = W_t - v_t$

Adaptive algos

- AdaGrad: learning rate adapted to the gradient $r_{t+1} = r_t + g_t \odot g_t$ and $W_{t+1} = W_t - \frac{\lambda}{\delta + \sqrt{r}} \odot g_t$
- RMSProp: exponentially weighted average $r_{t+1} = \rho r_t + (1-\rho) g_t \odot g_t$ and $W_{t+1} = W_t - \frac{\lambda}{\delta + \sqrt{r}} \odot g_t$
- Adam = RMSProp + Momentum

Initialization
Initialization can determine whether the algorithm converges or not and how quickly it does.

- Random or Sample from Gaussian/Uniform distribution.
- Xavier/Glorot Initialization: Gaussian with zero mean and 0.01 sd or $\mathrm{var}\left(W^{(l)}\right) = \frac{1}{\mathrm{Fan_{in}} + \mathrm{Fan_{out}}}$
- He-Normal Initialization: $\mathrm{var}\left(W^{(l)}\right) = \frac{2}{\mathrm{Fan_{in}}}$

IV. Normalization Methods

Batch Normalization: stabilize input distribution for each layer by setting mean and variance of each activation to be zero and one.
Formally, $\mu_B^{(k)} = \frac{1}{m} \sum_{i=1}^{m} x_i^{(k)}$ and $\sigma_B^{(k)} = \frac{1}{m} \sum_{i=1}^{m} \left(x_i^{(k)} - \mu_B^{(k)}\right)^2$ then $\hat{x}_i^{(k)} = \frac{x_i^{(k)} - \mu_B^{(k)}}{\sqrt{\sigma_B^{(k)^2} + \epsilon}}$ and $y_i^{(k)} = \gamma \hat{x}_i^{(k)} + \beta$
→ Improve Lipschitzness of the function so large gradient direction are less risky.

V. Model Compression

Why? Models are over parametrized $O(GB)$
Pruning: applying mask to weights(?)

Lecture 02: Convolutional Neural Networks

I. Introduction

A convnet contains convolutional filters (kernel), auxiliary layers to improve efficiency, feed forward layers.
Kernel size:

- How many filter outputs are influenced by a single input dimension.
- How many receptive units influence a single filter output.

II. Convolutions

Notation: input layer → convolution layer (affine transform) → detector layer (nonlinearity) → pooling layer
Input: L; kernel of size $k \times k$: K; output: O

Cross-correlation: $O(i,j) = (L * W)(i,j) = \sum_m \sum_n L(i+m, j+n) W(m,n)$

Strides: skip over some positions of the kernel to reduce computational cost.
Padding: pad the edges with extra fake pixels to produce an output with same size.
Channels: more than 2 dimensions → define a kernel per channel

Sizing Convolutional Filter

→ Input size: $h_L \times w_L \times d_L$
→ Filter params:

- Number of kernels: K
- Kernel size: $h_K \times w_K$
- Stride: s
- Padding: p

→ Output size: $\left(h_O = \frac{h_L - h_K + 2p}{s} + 1\right) \times \left(w_O = \frac{w_L - w_K + 2p}{s} + 1\right) \times (d_O = K)$
→ Total params: $\text{weights} = (h_K \times w_K \times d_L) \times K$ and $\text{biases} = K$.

Effective Implementation

1. Concatenate patches to form a single matrix P
2. Flatten the convolutional kernel to get a row vector $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \rightarrow (1 \quad 2 \quad 3 \quad 4)$
3. Perform a row-vector, matrix multiplication (get a row-vector with the same number of cols as P)
4. Reshape the results (h_O, w_O, d_O).

Pooling Layer

Reduce the size of feature maps to decrease the computational power required to train the network. Approximately invariant to translations and rotations.
Similar to convolutional filters, but nothing to learn here.

- Max Pooling Layer: keeps only the activation with the max value.
- Average Pooling Layer: averages the activations values within the filter mask.

Sizing Pooling Layers
→ Input size: $h_L \times w_L \times d_L$
→ Pooling Layer params:

- Kernel size: $h_K \times w_K$
- Stride: s

→ Output size: $\left(h_O = \frac{h_L - h_K}{s} + 1\right) \times \left(w_O = \frac{w_L - w_K}{s} + 1\right) \times (d_O = d_L)$
→ Total params: $\text{weights} = 0$ and $\text{biases} = 0$.

For the math: https://pdfs.semanticscholar.org/5d79/11c93ddcb34cac088d99bd0cae9124e5dcd1.pdf

III. Popular Architectures

Classic Blueprint: convolutions + activation (ReLU) + Maxpooling 2x2 + fully connected output + Softmax
AlexNet: data augmentation (x2048)
VGG-16: very deep convolutional net for large-scale image recognition (138M params)
ResNet: skip layers when they are not needed ($W = 0$)
Pre-Trained Models:

Training a model on ImageNet database takes weeks.
→ Transfer learning: use information from previously trained model
→ Fine tuning: retrain (some) params of the network

IV. Advanced Topics

Localization: predict the coordinates of a bounding box (x, y, w, h).
Evaluation metric: intersection over Union $\text{IoU} = \frac{\text{Area of Overlap}}{\text{Area of Union}}$ (ground-true and predicted bounding boxes)

Object Detection
Challenges: unknown number of objects; detection and classification tasks
Two-stage methods (region of interest then classification): slide windows (of different sizes) over the image and identify object using classification architectures (brute force method).
One-stage methods: detect and classify at the same time. E.g. YOLO: pre-trained on ImageNet and the loss is a weighted sum of classification, localization and confidence losses.

Segmentation
Semantic Segmentation: identify the object category of each pixel (cat vs. background)
Instance Segmentation: identify the object instance of each pixel (cat1 vs cat2)

(see examples)

Lecture 3: Sequence Modeling

I. Introduction

Sequential data (≠ i.i.d. data): successive points that are strongly correlated
Sequence Learning: ML algos used for sequential data:

- Time-series Prediction: use past values to predict future values (stock market, weather)
- Sequence Labeling: assign label for each member of a sequence (speech & handwriting recognition)

Methodologies:

- Auto-regressive models: predict the next term from a fixed number of previous terms.
- Feed-forward NN: generalize the previous model by using more layers and non-linear activations.

Sequence Labeling

This is a supervised learning task: use pairs $(x, t) \in \mathbb{R}^m \times L$ of input sequence and target label (m is fixed and L is the alphabet), to train an algo. Apply standard ML algos only at the level of sequences, not points.

Example: online handwritten recognition (the input is coordinates of the pen so $m = 2$)
The label L: the Latin alphabet, and extra labels for punctuation.
Error measure: edit distance between the output of the classifier and the target sequence (e.g. Levenshtein distance: minimum series of operations (add, replace, delete) to transform a into b).

II. Recurrent Neural Networks

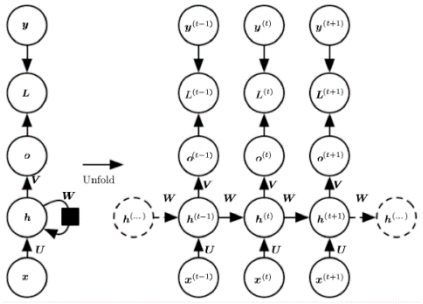
Preliminaries

An RNN is a model specialized for processing sequences of values $\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(\tau)}$ where τ is the sequence length ($t \in \{1, \dots, \tau\}$ is time or position in the sequence). It operates on mini-batches (e.g. sentences) of different τ's.
A dynamic system: $\boldsymbol{s}^{(t)} = f\left(\boldsymbol{s}^{(t-1)}; \theta\right)$
A dynamic system driven by external signal: $\boldsymbol{s}^{(t)} = f\left(\boldsymbol{s}^{(t-1)}, \boldsymbol{x}^{(t)}; \theta\right)$

RNN Formulation

General definition: input $\boldsymbol{x}^{(t)}$, hidden state $\boldsymbol{h}^{(t)}$, output $\boldsymbol{o}^{(t)}$, target $\boldsymbol{y}^{(t)}$, loss $\mathcal{L}\left(\boldsymbol{o}^{(t)}, \boldsymbol{y}^{(t)}\right)$
Parameters: input weights, hidden weights, and output weights ($\boldsymbol{U}_{i,j}$, $\boldsymbol{W}_{i,j}$ and $\boldsymbol{V}_{i,j}$ connecting unit i to j)

$$h^{(t)} = \tanh\left[Ux^{(t)} + Wh^{(t)} + b\right]$$
$$o^{(t)} = Vh^{(t)} + c$$
$$\hat{y}^{(t)} = \text{softmax}\left(o^{(t)}\right)$$



III. Training RNN

Defining the loss: $\mathcal{L}\left(\left\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(\tau)}\right\}, \left\{\boldsymbol{y}^{(1)}, \dots, \boldsymbol{y}^{(\tau)}\right\}\right) = \sum_t \mathcal{L}^{(t)} = -\sum_t \log p\left(y^{(t)} | \left\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(\tau)}\right\}\right)$
Backpropagation through time
Gradient w.r.t. the output: $\left(\nabla_{o^{(t)}}\mathcal{L}\right)_i = \frac{\partial \mathcal{L}}{\partial o_i^{(t)}} = \frac{\partial \mathcal{L}}{\partial \mathcal{L}^{(t)}} \frac{\partial \mathcal{L}^{(t)}}{\partial o_i^{(t)}} = \hat{y}_i^{(t)} - 1_{i,y^{(t)}}, \forall i, t$

Next, we work backpropagation starting from the end of the sequence:

- At $t = \tau$: $\nabla_{h^{(\tau)}}\mathcal{L} = V^T \nabla_{o^{(t)}}\mathcal{L}$ (chain rule)
- For $t < \tau$: $\nabla_{h^{(t)}}\mathcal{L} = W^T\left(\nabla_{h^{(t+1)}}\mathcal{L}\right).diag\left(1 - \left(h^{(t+1)}\right)^2\right) + V^T \nabla_{o^{(t)}}\mathcal{L}$ (no math details)

Gradient w.r.t. the weights:

- $\nabla_c \mathcal{L} = \sum_t \left(\frac{\partial o^{(t)}}{\partial c}\right)^T \nabla_{o^{(t)}}\mathcal{L} = \sum_t \nabla_{o^{(t)}}\mathcal{L}$
- $\nabla_b \mathcal{L} = \sum_t \left(\frac{\partial h^{(t)}}{\partial b^{(t)}}\right)^T \nabla_{h^{(t)}}\mathcal{L} = \sum_t diag\left(1 - \left(h^{(t)}\right)^2\right).\nabla_{h^{(t)}}\mathcal{L}$
- $\nabla_V \mathcal{L} = \sum_t \sum_i \left(\frac{\partial \mathcal{L}}{\partial o_i^{(t)}}\right)^T \nabla_V o_i^{(t)} = \sum_t \left(\nabla_{o^{(t)}}\mathcal{L}\right).h^{(t)^T}$
- $\nabla_W \mathcal{L} = \sum_t \sum_i \left(\frac{\partial \mathcal{L}}{\partial h_i^{(t)}}\right)^T \nabla_W h_i^{(t)} = \sum_t diag\left(1 - \left(h^{(t)}\right)^2\right).\left(\nabla_{h^{(t)}}\mathcal{L}\right).h^{(t-1)^T}$
- $\nabla_U \mathcal{L} = \sum_t \sum_i \left(\frac{\partial \mathcal{L}}{\partial h_i^{(t)}}\right)^T \nabla_U h_i^{(t)} = \sum_t diag\left(1 - \left(h^{(t)}\right)^2\right).\left(\nabla_{h^{(t)}}\mathcal{L}\right).x^{(t)^T}$

What is wrong? Exploding/Vanishing gradients.
Gradient clipping: if $\|g\| > v$ then $g \leftarrow \frac{g}{\|g\|}$

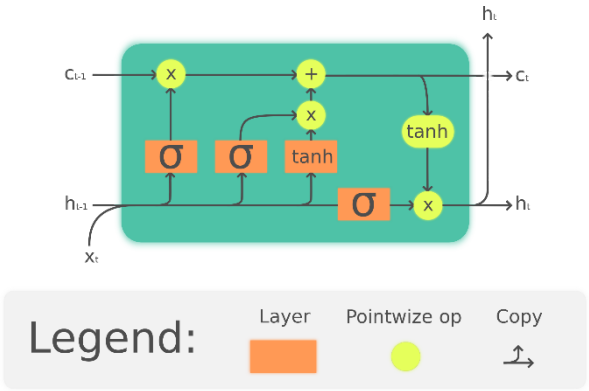
IV. Memory-based Architectures

RNN struggle to remember the past, as gradients vanish → design a cell to remember the past.
Long Short-Term Memory (LSTM): https://www.bioinf.jku.at/publications/older/2604.pdf

- The Forget Gate: $f_t = \sigma\big(W_f[h_{t-1}, x_t] + b_f\big)$ where $[.,.]$ means concatenation. → which information we want to delete from the cell state.
- The Input Gate: $i_t = \sigma(W_i[h_{t-1}, x_t] + b_i)$ which elements to update, and $\tilde{C}_t = \tanh(W_C[h_{t-1}, x_t] + b_C)$ which values to store.

→ Cell State update: $C_t = f_t \otimes C_{t-1} + i_t \otimes \tilde{C}_t$

- The Cell Output: $h_t = o_t \otimes \tanh C_t = \sigma(W_o[h_{t-1}, x_t] + b_o) \otimes \tanh C_t$



V. Beyond Recurrent Neural Networks

Problems with RNN: long-term dependency, vanishing gradients, computational cost.
Problems with LSTM: still challenging to remember long sequence, computationally heavy.
Both: not hardware-friendly, difficult to parallelize

Temporal Convolutional Network

- CNN with input sequence of any length
- No leakage from the future

Delated Convolutions
Residual Connections $o = activation\big(x + f(x)\big)$ to allow layers to learn about modifications.

Why TCN? Conv are parallel (no sequential), stable gradients, partial results are not stored.
Why not TCN? Not flexible for transfer learning usage since ≠ domains have ≠ requirements on history size.

Transformer Networks

Check videos about Transformer Networks! (https://youtu.be/iDulhoQ2pro)