A1: DEEP LEARNING

ML: step-by-step approach; DL: learn features/classification parameters jointly.

Linear Classifier —a straight line which best divide the space between classes. Linear combination $\hat{y} = g(w_0 + \sum x_i w_i)$ where w_0 is the bias, g is a non-linear function and $\hat{y}$ is the output. You may have many outputs so that $y_i = g(z_i), z_i = w_{0,i} + \sum x_j w_{j,i}$. You also may have many hidden layers so that the weights will be index: $z_{k,i} = w_{0,i}^{(k)} + \sum g(z_{k-1,j}) w_{j,i}^{(k)}$, i.e., $z_{k,i}$ values at a given layer are computed from the weighted activation outputs from the previous layer $z_{k-1,i}$ values.

Optimize weights according to a loss $\mathcal{L}(f(x^{(i)}; W), y^{(i)})$:

MSE: $L(W) = \frac{1}{n} \sum_{i=1}^{n} (y^{(i)} - f(x^{(i)}; W))^2$

BCE: $-\frac{1}{n} \sum_{i=1}^{n} y^{(i)} \log(f(x^{(i)}; W)) + (1 - y^{(i)}) \log(f(x^{(i)}; W))$

$W* = \underset{W}{\operatorname{argmin}} \frac{1}{n} \sum \mathcal{L}(f(x^{(i)}; W), y^{(i)})$

GD: (1) random initial weights; (2) compute $\frac{\partial L(W)}{\partial W}$; (3) small step in the other direction $W = W - \alpha \frac{\partial L(W)}{\partial W}$. Repeat until convergence.

Backpropagation to compute the gradient: $\frac{\partial L(W)}{\partial w_i} = \frac{\partial L(W)}{\partial \hat{y}_i} \cdot \frac{\hat{y}_i}{\partial z_{i+1}} \cdot \frac{\partial z_{i+1}}{\partial w_i}$.

Rule: local gradient $\times$ upstream gradient.

If the derivative of the activation function is too small, the gradient is too small and the model does not learn.

CNN — neurons are connected to part of the neurons in previous layer; neurons in a single layer function share weights; those connections slide over the neurons of the previous layer $\therefore$ high decrease in # trainable params. Convolution is to linearly apply a filter to the image. This filter (kernel) is a matrix of weights.

Pooling — replaces the output of the net at a certain location with a summary statistic of the nearby outputs. Useful for downsampling. Convolution $\rightarrow$ Activation function $\rightarrow$ Pooling.

Activation Functions

Sigmoid $\frac{1}{1+e^{-x}}; \sigma(x)(1 - \sigma(x))$ — saturated neurons 'kill' the gradients; outputs are not zero-centered; `exp()` is expensive.

Tanh $\frac{e^x - e^{-x}}{e^x + e^{-x}}; 1 - tanh(x)^2$ — kills the gradients when saturated.

ReLU $\max(0, x); 0$ if $x < 0, 1$ if $x \geq 0$ — efficient; converges faster; not zero-centered; kills gradient for negative inputs.

LeakyReLU $\max(0.01x, x); 0.01$ if $x < 0, 1$ if $x \geq 0$ — never saturates; efficient; push the mean of activations closer to zero; not zero-centered output.

ELU $\alpha(e^x - 1)$ if $x < 0, x$ if $x \geq 0; ELU(x) + \alpha$ if $x < 0, 1$ if $x \geq 0$ — pros of LeakyReLU, but saturates at smaller values $\therefore$ more robust to noise; not zero-centered output.

Preprocessing: Usual practices of preprocessing operations are: zero center; subtract the mean image; subtract per-channel mean; normalize from -1 to 1.

Small initial weights: vanishing gradient; Large initial weights: depending on the activation function, the input will be in the saturated region (small gradient), leading to the same problem.

Batch Normalization $\hat{x}^{(d)} = \frac{x^{(d)} - \mathbf{E}[x^{(d)}]}{\sqrt{\mathrm{Var}[x^{(d)}]}}$ — smooths loss function. Makes the best use of nonlinear funcs, which happens around zero.

Undoing normalization: $y^{(d)} = \gamma^{(d)} \hat{x}^{(d)} + \beta^{(d)}$, where $\gamma^{(d)}$ is the variance and $\beta^{(d)}$ is the mean. Reduces the dependence on init w & allows higher lr.

Optimizers

SGD — $g = \nabla_{\theta_d} \sum \frac{1}{m} L[f(w, x_i), y_i] - \lambda R(w)$; then $w = w - \alpha g$. It has no memory. (1) slow progress along shallow dimension, jitter along steep direction; (2) may get stuck in local minima/saddle point.

SGD + Momentum — $v_k = p v_{k-1} - \alpha \nabla L(w_k); w_{k+1} = w_k + v_k$. Extrapolates with some memory of the past.

Nesterov Momentum — $v_k = p v_{k-1} - \alpha \nabla L(w_k + \alpha p_{k-1}); w_{k+1} = w_k + v_k$. Computes the gradient of the extrapolated.

AdaGrad — decreases learning rate as approaches the minimum. $r_t = \sum (\frac{\partial L}{\partial w_{t-1}})^2; w_t = w_{t-1} - \frac{\alpha}{\sqrt{r_t} + \epsilon} \odot \nabla L(w)$.

RMSProp — turns the gradient accumulation into a weighted moving average. $r = \delta \cdot r + (1 - \delta) \cdot \nabla L(w) \odot L(w)$.

Adam — $v = \rho_1 v - (1 - \rho_1) \nabla L(w)$; $r = \rho_2 \cdot r + (1 - \rho_2) \cdot \nabla L(w) \odot \nabla L(w)$. Then $v = \frac{v}{1 - \rho_1^t}, r = \frac{r}{1 - \rho_2^t}$ and $w = w - \alpha \frac{v}{\sqrt{r} + \epsilon}$. It combines momentum with adaptative lr.

Better generalizing: **(1) early stopping** if loss on valid set does not improve from the best loss in a range; **(2) model ensembles**; **(3) regularization**, $L(W) = \frac{1}{n} \sum \mathcal{L}(f(x^{(i)}; W), y^{(i)}) + \lambda R(W)$. In L2 (weight decay), $R(W) = \sum_k \sum_i w_{k,l}^2$; **(4) dropout**, randomly turning off nodes, forcing the network to learn redundant representations (learns not to trust in specific nodes). Somewhat a large ensemble of models; **(5) data augmentation** adding new noisy samples (mirror, rotate, color jittering...); **(6) transfer learning** using a pre-trained network, freeze earlier layers and retrain final layers (fine tuning).

CNN Architectures

LeNet — 3 CONV, 2 AVGPOOL and 2 FC. Uses tahn. Mainly for classifing digits (MNIST).

AlexNet — 5 CONV, 3 MAXPOOL, 2 NORM, 3 FC. Uses ReLU (faster) & dropout before FC6 and FC7. Many parameters to train.

ZFnet — improves AlexNet hyperparameters (stride, filters, normalization). Uses Local Contrast Normalization, enforcing local competition between adjacent features in a feature map, and between features at the same spatial location in different feature maps.

VGGNet — 3x3 CONVs and 2x2 MAXPOOL, 3 FC followed by a softmax. Used as feature extractor (backbone) and image classification (fine-tuning).

GoogLeNet — pushes speed & acc, not by stacking layers. Adds modules of Conv $\rightarrow$ Pool $\rightarrow$ Softmax $\rightarrow$ Concat/Norm. The conv layers have different shapes to better capture global & local information. Adds 1x1 convolutions to decrease # params. It has auxiliar classifiers, and total loss is a weighted sum of auxiliary and main loss.

ResNet — proposes residual blocks: use network layers to fit a residual mapping, i.e., instead of learning $H(x)$, the convolutions learn what must be added or substracted from the input to generate $H(x)$. $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial H} \frac{\partial H}{\partial x} = \frac{\partial L}{\partial H} (\frac{\partial F}{\partial x} + 1) = \frac{\partial L}{\partial H} \frac{\partial F}{\partial x} + \frac{\partial L}{\partial H}$, where $H(x) = F(x) + x$.

DenseNet — each layer is connected to every other layer. Alleviates vanishing gradient, strengthens feature propagation and encourages feature reuse.

Transfer Learning: use outputs or layer(s) of a trained network on a different task as generic feature detectors. Cut off top

layer(s) and train new one(s) with supervised objective for target domain (fine-tuning).

(1) large dataset diff from pre-trained's → train from scratch; (2) small dataset diff from pre-trained's or (3) large dataset similar to pre-trained's → train some layers and freeze others; (4) small dataset similar to pre-trained's → freeze conv base.

Semantic Segmentation: identifies the object category of each pixel (labels are class aware). Metrics: Intersection over Union & Precision $= \frac{TP}{TP+FP}$. Fully Convolutional – downsampling and upsampling. One can upsample by repeating the entry; putting it in some block; or memorizing where the first entry came from.

SegNet — MaxPooling indices to upsample. Learnable Upsampling: Transpose Convolution. Output contains copies of the filter weighted by the input, summing up at overlaps in the output.

U-Net — skip connections: output of corresponding layer in contractive stage is appended to the inputs of the expansive stage.

ResUNet — replaces UNet neural units by residual blocks to ease training.

DeepLabv1 & DeepLabv2 — Atrous Conv. inserts $r-1$ zeros between consecutive filter values along each dimension. Avoids loosing location/spatial information.

PARSENet & PSPNet — global context helps clarifying local confusion; Image Pooling or Image Level Feature.

DeepLabv3 & DeepLabv3+ — combines atrous convolution and skip connections to recover object boundaries information lost due to pooling and striding, at a lower computational cost.

Loss Functions: (1) CE $= -\frac{1}{N}\sum \log(p_i)$, where p_i is the probability of the true class at site i and N is the number of samples being classified. Majority dominates and minority tends to vanish; (2) BCE: $-\frac{1}{N}\sum w_{y_i}\log(p_i)$, where w_{y_i} is the weight for the true class; (3) FL $= -\frac{1}{N}\sum(1-p_i)^\gamma \log(p_i)$; (4) BFL $= -\frac{1}{N}\sum w_{y_i}(1-p_i)^\gamma \log(p_i)$; (5) L_2 or L_1 when the output is non-categorical (regression).

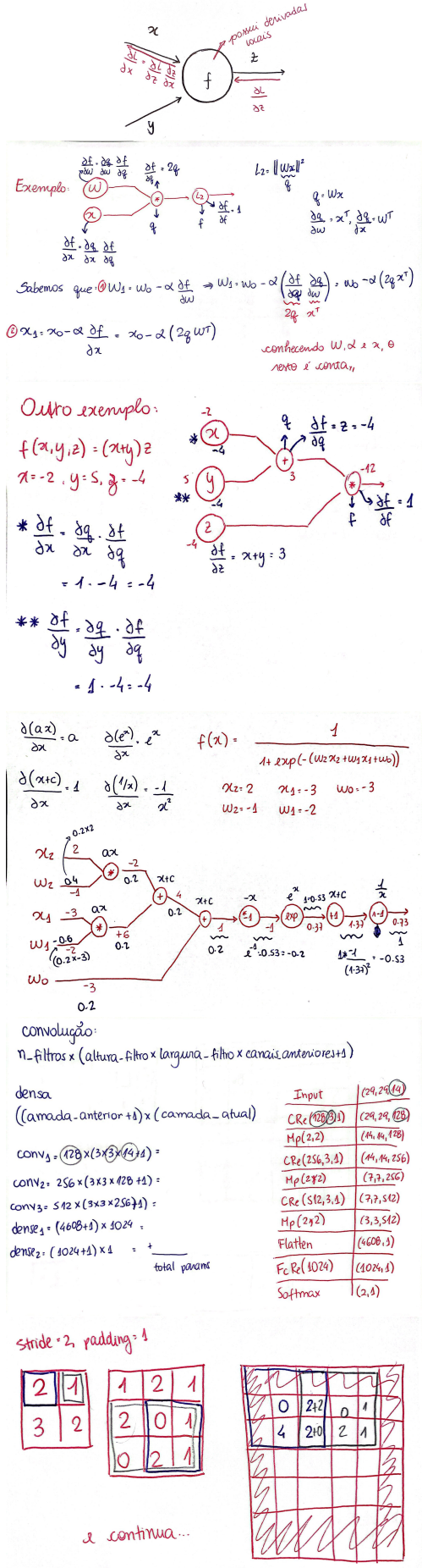
Object Detection: locates multiple instances of object classes with bounding boxes. Has to do with image classification and object localization. Performance metrics are IoU, Precision $= \frac{TP}{TP+FP}$, Recall $= \frac{TP}{TP+TN}$. Mean average precision (mAP) $= \frac{\sum_{i=1}^{K} AP_i}{K}$, which is the mean of AP across all K classes.

YOLO: train labels contains the bounding box coordinates for each grid cell; indicates if there is an object in each grid cell; and from which class. Object is assigned to the grid cell that contains its midpoint. $y = 3\times 3\times(5+\#\text{ classes})$. (b_x, b_y) is the position of the midpoint of the object; (b_h, b_w) is the size of the bounding box of the object. With two overlapping objects in the same grid cell, $y = 3\times 3\times(2\times(5+\#\text{ classes}))$. Keep only the boxes with the largest p_c, which is the probability of the that thing belong to the class.

Recurrent Neural Networks: modeling sequencial data. Speech recognition, language translation, image captioning, multitemporal image analysis, video analysis...

Process a sequence of vectors x_t by applying a recurrence formula at every time step: $h_t = f_\theta(h_{t-1}, x_t)$, where f_θ is some function with parameters θ and x_t is the input vector at the current step. Suppose $T=3$, so $h_3 = f_\theta(h_2, x_3) = f_\theta(f_\theta(h_1, x_2), x_3) = f_\theta(f_\theta(f_\theta(h_0, x_1), x_2), x_3)$.

BTT: go through RNN calculating loss for all timesteps and updates the weights. Truncate BPTT avoids vanishing or exploding gradient updating only the last k steps.



L_1 is somewhat a resource selection because it can eliminate weights with less important characteristics.

k-fold Cross validation: we divide the data into k-groups. One of them is the test set and the others are training, repeating k times until all of them are used as test once. It is expensive, but allows all data to be used as training.