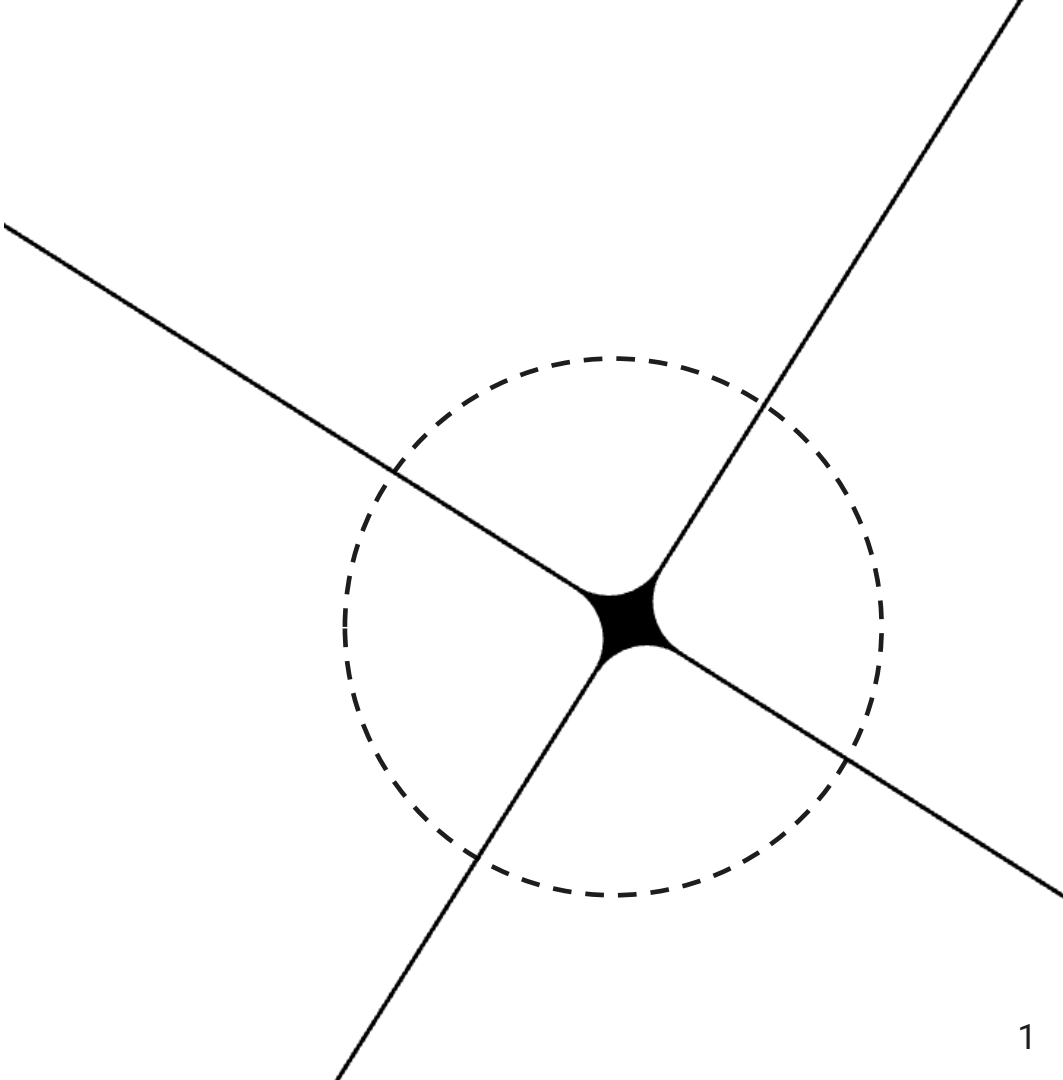
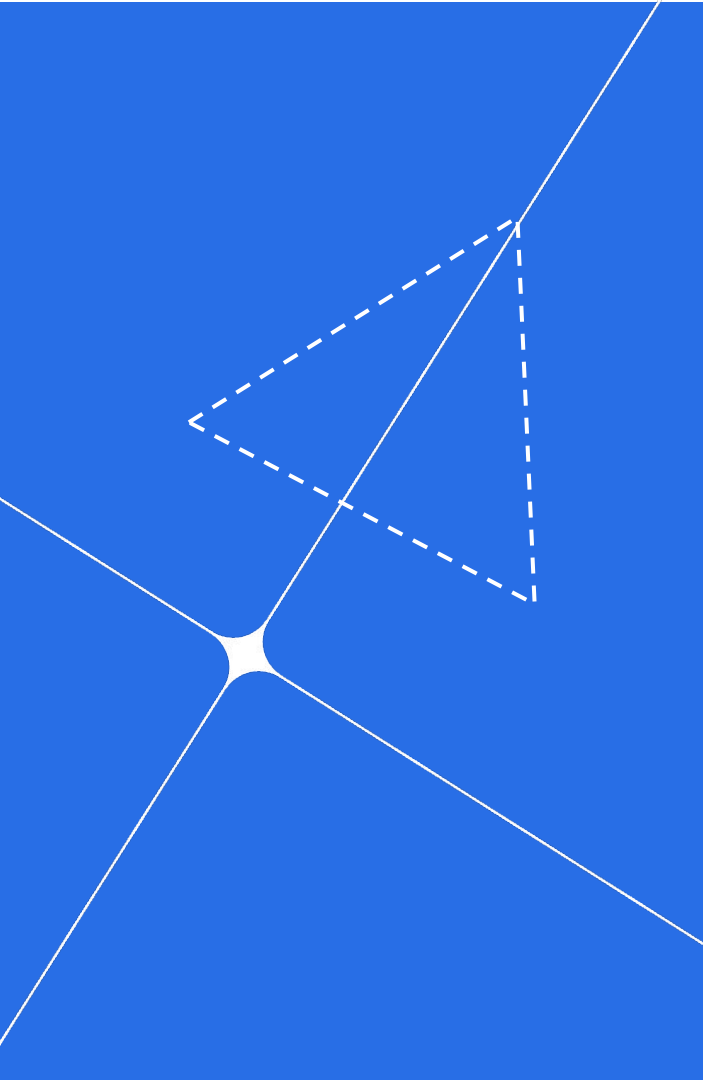
INSTITUT POLYTECHNIQUE DE PARIS

TELECOM SudParis

Modern Computer Vision

Julien Romero



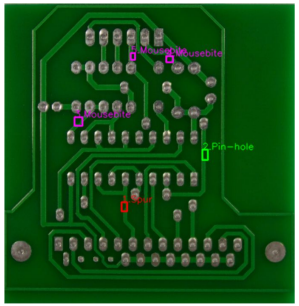


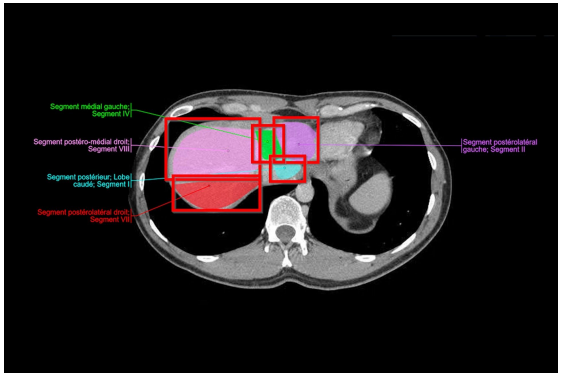
Introduction et motivation

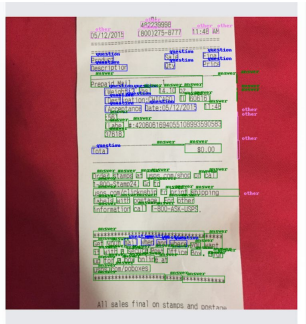
Pourquoi la vision “compte” en 2026

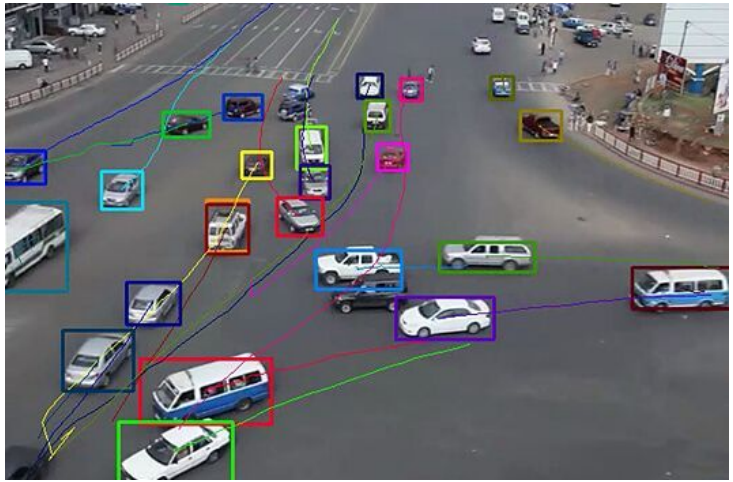
- Les capteurs (caméras, vidéos, scanners) sont partout : industrie, retail, mobilité, santé, documents
- La valeur business est souvent directe : **réduction coûts** (contrôle qualité, back-office), **augmentation revenus** (personnalisation, UX), **réduction risques** (sécurité)
- Les modèles modernes ont changé la donne : **pré-entraînement massif** + **fine-tuning léger** + parfois **zero-shot**
- En pratique, le problème n’est pas “juste” le modèle : c’est **data + pipeline + contraintes prod**
- Contraintes typiques en entreprise : latence, robustesse (lumière, angles), long tail, dérive (drift), RGPD/PI
- Coût caché majeur : **annotation** (temps humain) + **définition du label** (taxonomie, ambiguïtés)
- Objectif du cours : passer de “ça marche en notebook” à “ça marche dans un produit”

Exemples


Mousebite
Mousebite
Mousebite
Pin-hole
Pin-hole

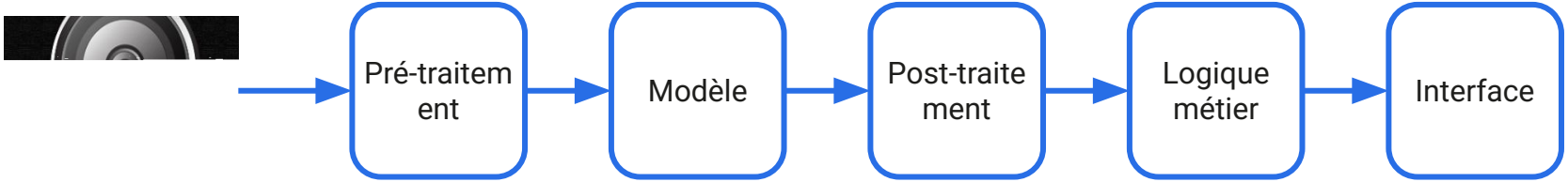

Segment médial gauche; Segment IV
Segment postéro-médial droit; Segment VIII
Segment postérieur; Lobe caudé; Segment I
Segment postérolatéral droit; Segment VII
Segment postérolatéral gauche; Segment II





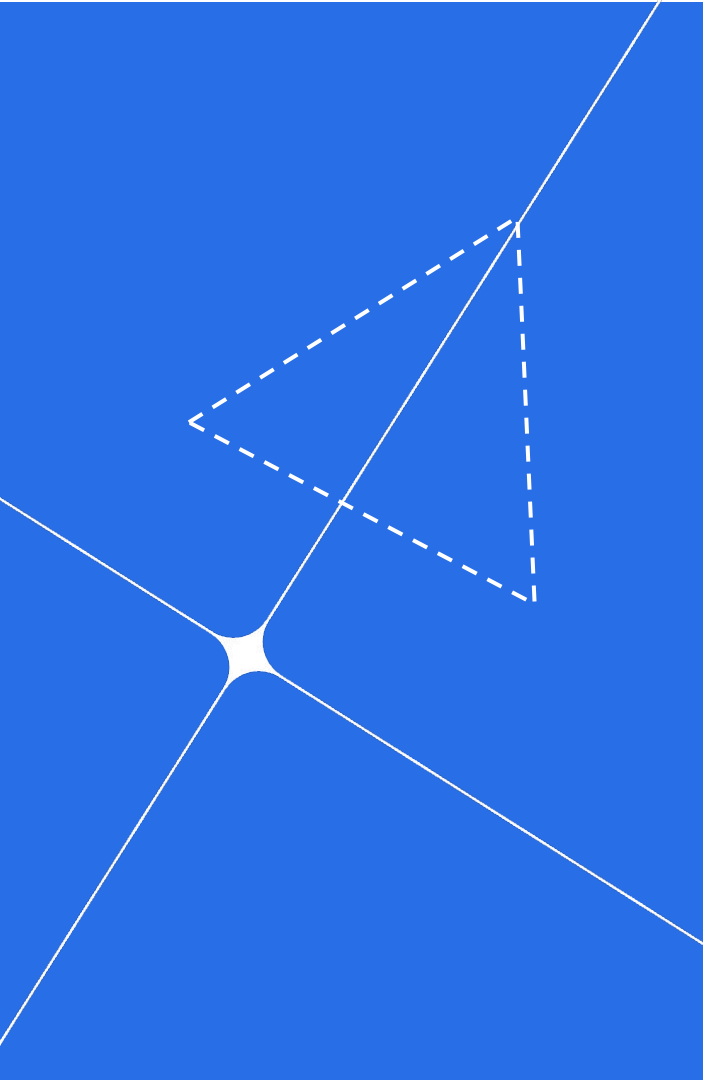
Définition opérationnelle de “Modern Computer Vision”

- **Computer Vision** : apprendre une fonction f_θ qui mappe une observation visuelle xxx vers une sortie yyy utile (classe, boîtes, masques, texte, trajectoires…)
- Spécificité moderne : **architectures** (ViT / CNN modernes / hybrides) + **foundation models** (ex : segmentation promptable)
- Représentations : embeddings visuels réutilisables (transfer learning), parfois alignés texte–image (multimodal)
- Paradigme dominant : **pré-entraînement** sur grands corpus pour adaptation à une tâche via fine-tuning / adapters / prompt
- “Dense prediction” vs “global” :
  - global : classification (1 label/image)
  - dense : détection/segmentation (structure spatiale, multi-objets)
- Les erreurs importantes en prod ne sont pas “moyennes” : elles sont dans les **cas rares** (long tail) et les **changements de contexte**
- Fil rouge : choix de la tâche + métrique + données + modèle + intégration impacte la performance produit



Le vrai problème : optimiser un système, pas un score de benchmark

- Objectif prod : maximiser une utilité U sous contraintes (latence, coût, risque)
  - Exemple de forme : $U = \text{Gain} - \lambda \cdot \text{Coût} - \mu \cdot \text{Risque}$
- Différence critique : benchmark = distribution “propre”, prod = distribution vivante
- Data-centric reality : labels incomplets/incohérents, classes mal définies, domain shift (caméra, site, saison)
- Robustesse : illumination, motion blur, occlusions, arrière-plans, nouveaux objets, adversarial “naturel” (reflets, écrans)
- Choix d’architecture = compromis :
  - accuracy vs latence vs mémoire vs facilité de déploiement
- Les trois questions de base avant tout entraînement :
  - quelle tâche exacte ? quelle métrique ? quel coût d’erreur (FP/FN) ?
- Résultat attendu à la fin de la séance : savoir “designer” un pipeline CV moderne crédible pour une équipe produit



Tâches, datasets et métriques

Pourquoi commencer par tâches + métriques (avant les architectures)

- En entreprise, on n'optimise pas "un modèle", on optimise une **décision** (accept/reject, alerte, extraction, tri)
- La tâche fixe la **sortie** yyy, donc le design du pipeline (pré/post-traitement, UI, stockage)
- La métrique doit refléter le **coût d'erreur** (FP vs FN) et le **niveau de granularité** (image / objet / pixel)
- Exemple :
  - contrôle qualité : une FN (défaut non vu) coûte plus cher qu'une FP (recontrôle humain)
  - OCR facture : la qualité "globale" importe moins que la qualité **des champs clés** (total, TVA)
- Risque majeur : "metric gaming" (bon score, mauvais produit) si la métrique est mal choisie
- Objectif : savoir écrire un *spec* clair : tâche → données → métriques → seuils → protocole d'éval

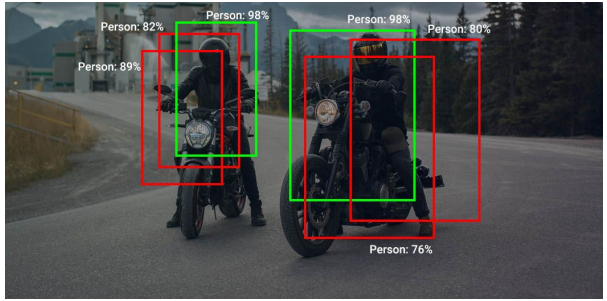
Use-case	Tâche	Métriques principales
Inspection	segmentation	mAP/IoU
Retail comptage	Détection + tracking	MOTA/IDF1
Lecture document	OCR	CER/WER + field acc.
Médical	segmentation	DICE/IoU
Sécurité	classification	ROC-AUC
QA produit	anomaly	PR-AUC

Classification et retrieval

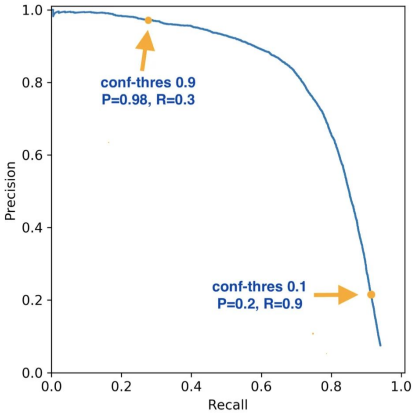
- **Single-label** : 1 classe/image
  - ex : type de défaut, espèce, catégorie produit
- **Multi-label** : plusieurs tags/image
  - ex : “rayure” + “tache” + “logo manquant”
- Métriques courantes :
  - Accuracy, Top-k, Precision/Recall/F1, ROC-AUC (classe rare), PR-AUC (fort déséquilibre)
  - En multi-label : seuil par classe, micro/macro-averaging (impact fort sur interprétation)
- *Calibration* utile : probas fiables => réglage de seuils et tri humain efficace
- Retrieval :
  - apprendre un embedding $z=f_\theta(x)$
  - similarité cosine
  - KPI = Recall@K / mAP retrieval
- Cas réel :
  - “trouver des produits similaires” = embeddings + ANN (FAISS/Qdrant/Chroma) + règles métier

Détection d'objets : sorties et difficultés réelles

- Sortie : ensemble $\{(b_i,c_i,s_i)\}_{i=1..N}$ (boîte, classe, score), N variable
  - La boîte contient l'objet détecté
  - $b_i=(x_1, y_1, x_2, y_2)$
- Problèmes concrets : petits objets, occlusions, objets collés, arrière-plans "trompeurs"
- Post-traitement classique :
  - seuil score
  - **NMS** (Non Max Suppression, suppression doublons)
  - choix seuil = trade-off FP/FN
- Annotation plus coûteuse que classification
  - boîtes = temps humain
  - ambiguïtés (bord exact ? objets partiels ?)
- Datasets typiques : COCO (généraliste), KITTI/BDD100K (conduite), SKU-110K (densité retail), etc.
- Usage industriel fréquent : "détecter puis…" (compter, suivre, segmenter, recadrer pour OCR)
- Pipeline mental : détecter = "où" + "quoi", mais pas "forme" (c'est la segmentation)



IoU, Precision–Recall, AP et mAP (détection)



- Overlap boîte-vérité : **IoU**
  - IoU(B,B’)=|B∩B’| / |B∪B’|
- Définition “match” : une prédiction est TP si IoU ≥ τ (ex : 0.5) et bonne classe
- En balayant le seuil de score, on obtient la courbe **Precision–Recall** (PR)
  - utile quand classes rares
- **AP** = average precision = aire sous la courbe PR (par classe), similaire à AUC (différentes interpolations)
- **mAP** = mean AP = moyenne des AP des classes
- COCO-style : mAP moyenné sur $\tau \in \{0.50, 0.55, \ldots, 0.95\}$ (plus strict, plus informatif)
- Lecture produit :
  - mAP haut mais recall faible, alors “on rate des objets”
  - mAP bas mais recall haut, alors “trop de FP”
- À retenir : mAP dépend fortement de la qualité des labels et de la définition de ce qu’est un objet

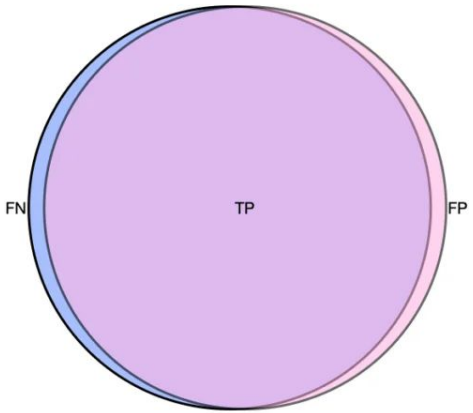
Segmentation : sémantique, instance, panoptique + métriques

- **Sémantique** : label/pixel (route, ciel)
- **Instance** : objet/pixel (chaque voiture)
- **Panoptique** : mix sémantique + instance
- Sortie
  - masque $M \in \{0,1\}^{H \times W}$ (binaire) ou $M \in \{1..K\}^{H \times W}$ (multi-classes)
- Métrique pixel
  - **IoU** par classe, moyenne (mIoU) ; sensible aux frontières et aux petites classes
- **Dice** (très utilisé quand objets petits / déséquilibre, ressemble F1)
  - $\text{Dice}(M,M') = 2|M \cap M'| \ / \ (|M| + |M'|)$
- Pourquoi ça compte : un modèle peut “bien classifier” mais rater les contours = mauvais pour mesure/surface
- Exemples
  - médical (organes/lésions)
  - industrie (défauts surfaciques), e-commerce (détourage produit)
- Pont vers la suite : segmentation moderne = modèles dédiés + **modèles promptables** (SAM)

Segmentation : sémantique, instance, panoptique + métriques

- **Sémantique** : label/pixel
- **Instance** : objet/pixel (ch
- **Panoptique** : mix sémant
- Sortie
  - masque M∈{0,1}
- Métrique pixel
  - **IoU** par classe, m
- **Dice** (très utilisé quand c
  - Dice(M,M')=2|M(
- Pourquoi ça compte : un
- Exemples
  - médical (organes
  - industrie (défauts
- Pont vers la suite : segm

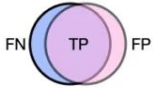
$FN = FP = 10$



for $TP = 1000$,

$DSC = 0.99$

$IoU = 0.98$



ure/surface

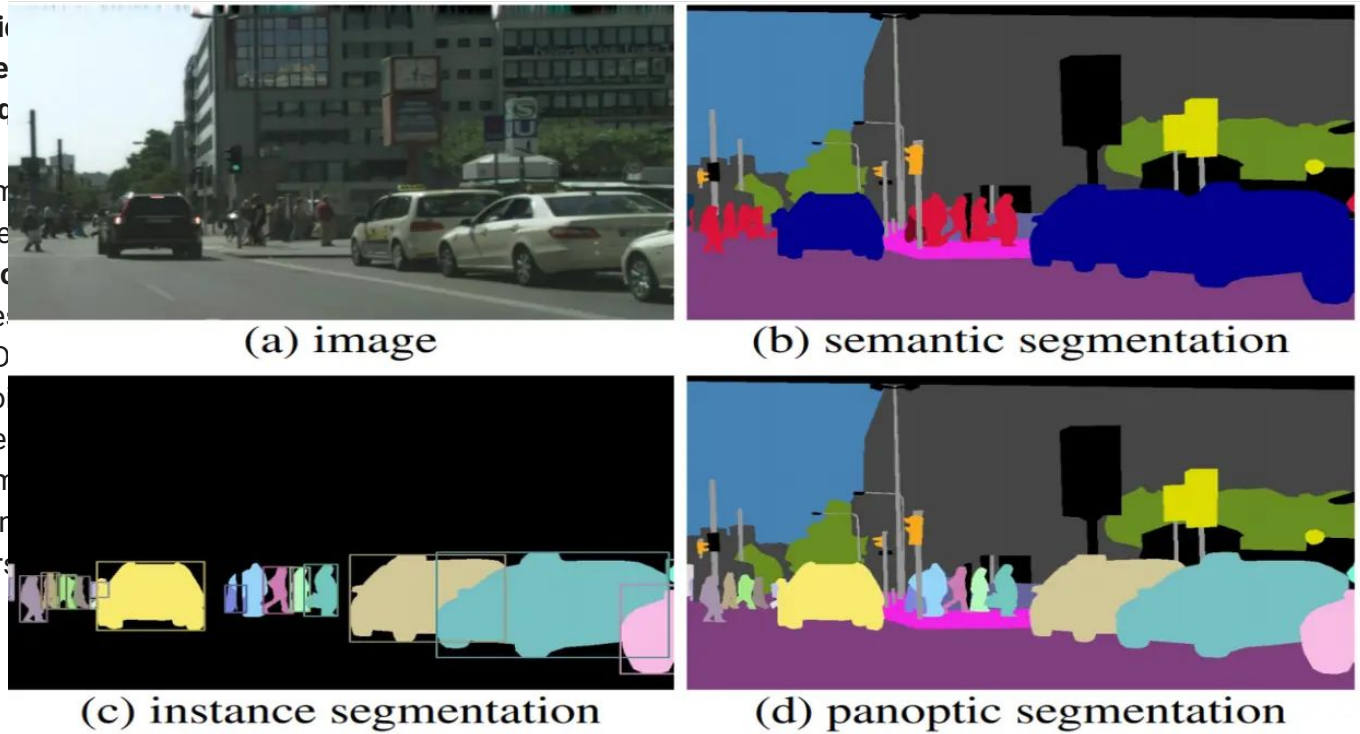
for $TP = 100$,

$DSC = 0.91$

$IoU = 0.83$

Segmentation : sémantique, instance, panoptique + métriques

- **Sémanti**
- **Instance**
- **Panoptic**
- Sortie
  - m
- Métrique
  - **Io**
- **Dice** (trè
  - D
- Pourquo
- Exemple
  - m
  - in
- Pont vers

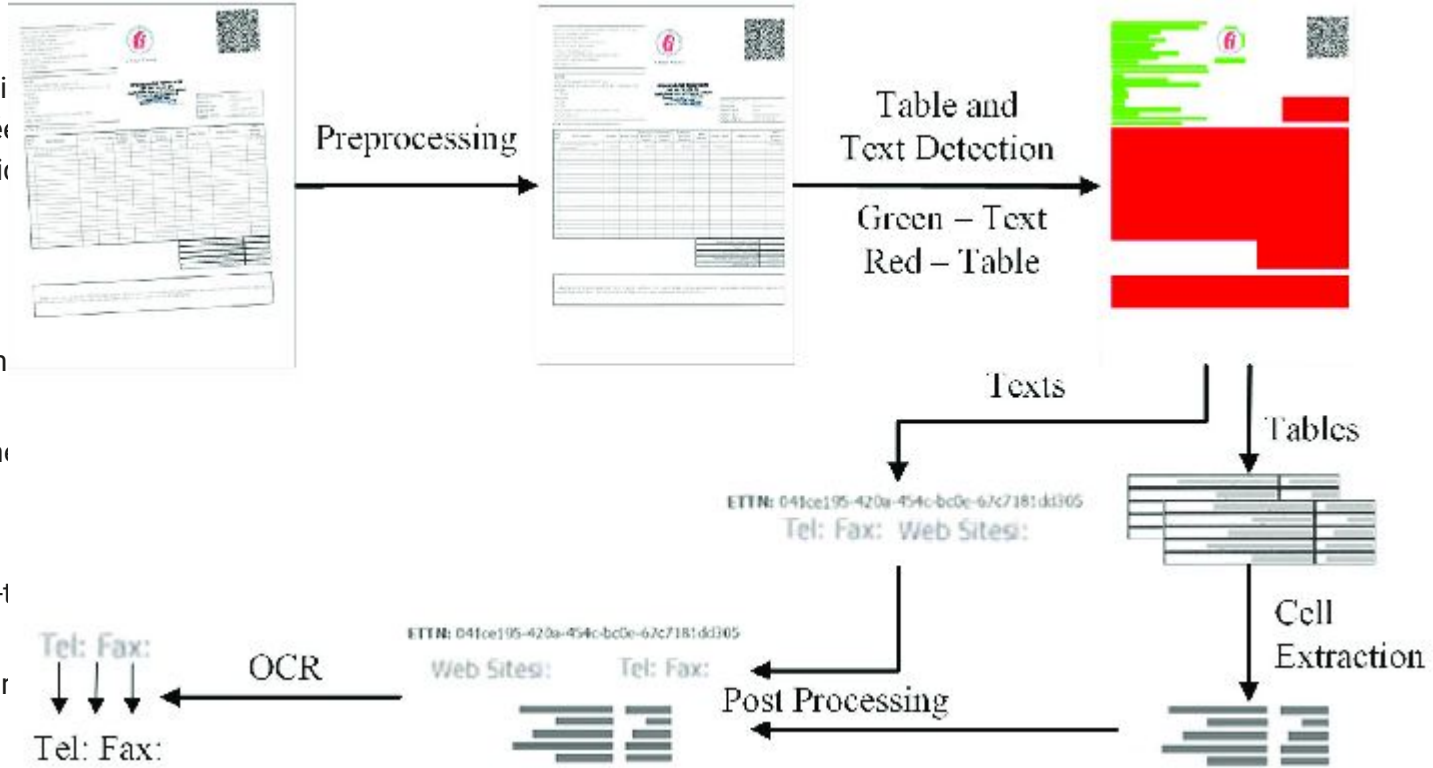


OCR / Document AI : quand la vision sert le texte et la structure

- Pipeline typique : **détection** zones texte → **reconnaissance** séquence → **structuration** (champs)
- Entrée variée : scan, photo smartphone, pdf rasterisé ; défis : perspective, blur, polices, tampon, handwritten
- Métriques OCR
  - **CER** (char error rate)
  - **WER** (word error rate)
  - mais KPI métier = exactitude champs
- Exemple KPI
  - “Total TTC exact” / “Date exacte” / “IBAN exact” (tolérance zéro possible)
- Données
  - annotations coûteuses (boîtes lignes/mots + transcription)
  - bruit de ground truth fréquent
- Post-traitement indispensable
  - regex, validation (TVA, SIRET), contraintes (montants, formats)
- Intégration : human-in-the-loop (validation) + journalisation des cas ambigus

OCR / Document AI : quand la vision sert le texte et la structure

- Pipeli
- Entré
- Métri
  -
  -
  -
- Exem
  -
- Donn
  -
  -
- Post-t
  -
- Intégr

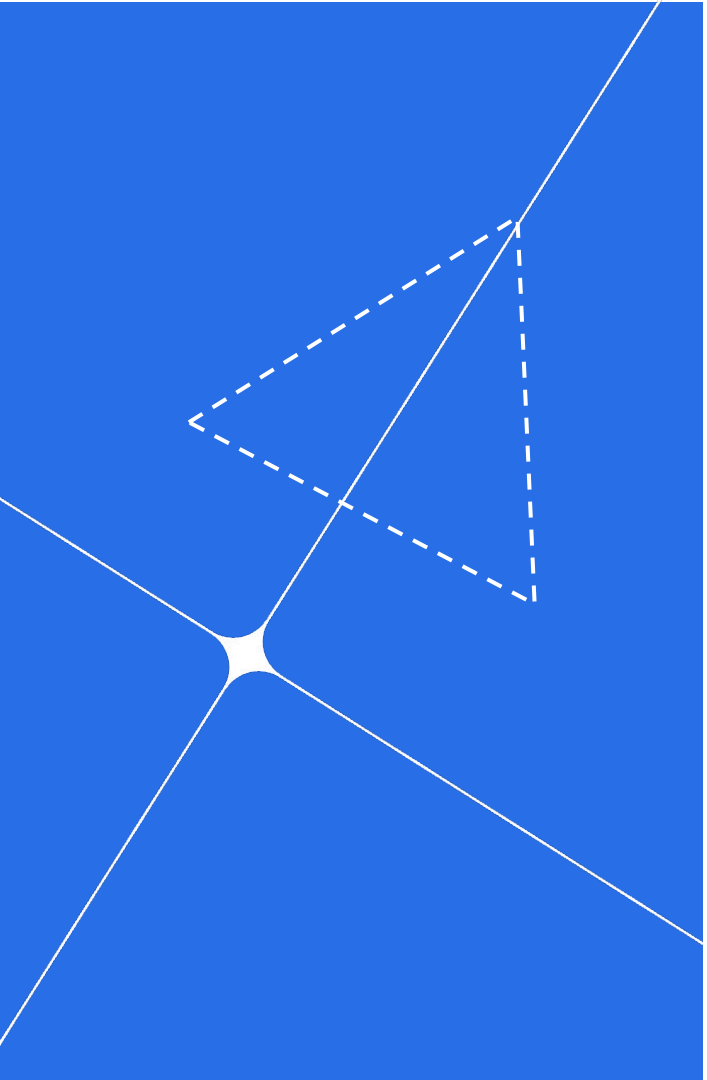


Tracking vidéo : suivre des identités

- Objectif : associer des détections au fil du temps avec trajectoires + identités stables
- Approche dominante : **tracking-by-detection** (détection par frame + association)
- Difficultés
  - occlusions, croisements, entrées/sorties, changements d'apparence, motion blur
- Intuition des métriques de Multiple object tracking (MOT)
  - **ID switches** (erreurs d'identité), **IDF1** (qualité identité), **MOTA/HOTA** (global)
- Sorties utiles
  - vitesse/trajectoire, comptage unique, temps passé dans une zone (analytics retail)
- Engineering
  - latence frame-to-frame, buffering, horodatage, synchronisation caméras (multi-cam)
- Bon réflexe : distinguer "erreur de détection" vs "erreur d'association" (debug séparé)

Choisir un dataset et un protocole d'évaluation

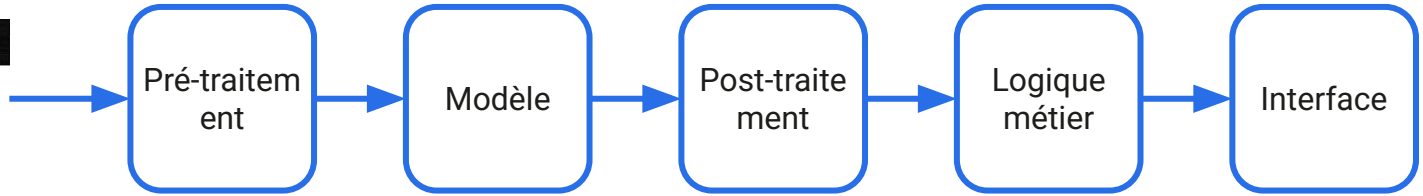
- Dataset public = bon départ, mais attention **domain gap** (capteur, angle, environnement, classes)
- Split correct : train/val/test sans fuite (mêmes scènes, mêmes objets, mêmes documents proches)
- Définir la taxonomie
  - classes, "autres", seuil d'ambiguïté, règles d'annotation (guidelines)
- Évaluer par *slices*
  - taille objet, luminosité, type de fond, rareté
  - But : trouver le "long tail"
- Mettre une baseline forte
  - modèle pré-entraîné + fine-tuning minimal + seuils calibrés
- Fixer des *acceptance criteria* produit
  - métriques + latence + mémoire + taux d'échec + couverture
- Préparer la prod : logging (entrées + sorties), jeux de tests "vivants", et plan de ré-annotation



Du pixel au système : représentations, pipeline et “points de rupture”

Pourquoi le pipeline compte plus qu'on ne le croit

- En prod, 50% des bugs CV viennent de l'**I/O** (décodage, formats, couleurs, tailles), pas du modèle
- Même modèle + preprocessing différent donne des outputs différents (ex : RGB/BGR, range [0,255] vs [0,1])
- La "bonne" représentation dépend de la tâche
  - global (classification) vs dense (detection/segmentation) vs embedding (retrieval)
- Le pipeline complet = **prétraitement → modèle → post-traitement → décision métier**
- Objectif : rendre chaque étape **testable**, **débogable**, et **contrôlable** (seuils, règles, logs)
- Point clé
  - séparer *model output* (probabilités, logits, masks) de *product decision* (alerte, extraction)
- On veut un système stable face aux variations "réelles" : capteur, lumière, compression, mouvement



Prétraitement : ce qui casse (souvent) en entreprise

- Décodage : JPEG/PNG, EXIF orientation, profils couleur, gamma (différences PIL/OpenCV)
- Canaux : **BGR vs RGB**, ordre des canaux, alpha channel (RGBA), niveaux de gris (1 canal)
- Échelles : uint8 [0..255] vs float [0..1], normalisation par canal (mean/std)
- Redimensionnement : conserver ratio (letterbox/pad) vs stretch ; impact sur détection/segmentation
- Crops/tiles : utile pour très grandes images (inspection, doc), mais impose une logique d'assemblage
- Augmentations : ok en train, **déterministes en eval** (sinon métriques instables)
- Reproductibilité : fixer seed, versionner transforms, tracer paramètres (pour audit/debug)

Abstraction “backbone + head” : lire un modèle comme une API

- Vue système :
  - y=head(backbone(x))
- Backbone : extrait des features F (maps multi-échelle pour détection/segmentation, ou embedding pour retrieval)
- Head : transforme F en sorties task-specific (logits, boxes, masks, keypoints, texte)
- Dense prediction : sorties structurées (spatialement) ⇒ importance du multi-scale (petits objets)
- Embeddings : même backbone peut servir classification *et* retrieval (ex : “similarity search”)
- Conséquence produit : on peut “swap” un backbone sans casser l’API de la head (si interfaces stables)
- Bon réflexe : documenter I/O du modèle (shape, dtype, conventions coordonnées, num classes)

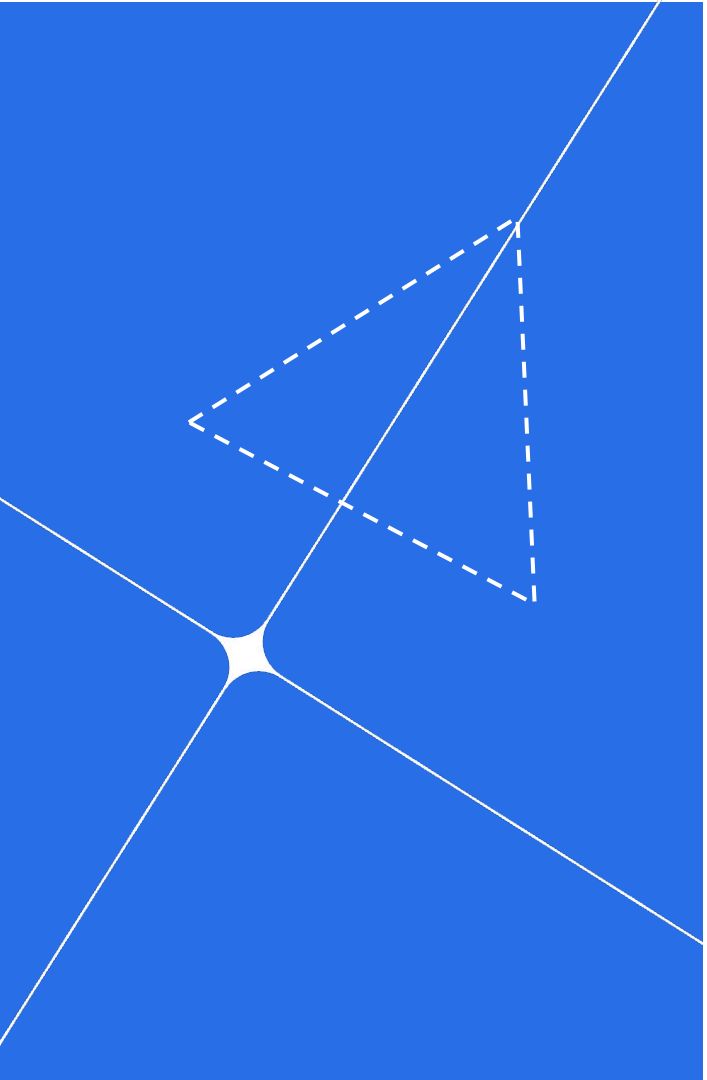
Post-traitement : transformer une sortie réseau en objet exploitable

- Détection : thresholding + NMS (ou équivalent) + conversion coordonnées = pixels image d'origine
- Segmentation : threshold mask, puis nettoyage (composantes connexes, remplissage trous) si besoin métier
- Calibration : un score 0.9 n'est pas toujours une "proba fiable"
  - Étude impact sur seuils et tri humain
- Règles métier
  - filtres (taille min/max, ROI)
  - agrégation (compter, mesurer, alerter)
- Formats de sortie : JSON "stable" (classes, scores, polygones/RLE, metadata) pour intégration produit
- Logs minimaux : version modèle, params preprocessing, seuils, top-K sorties + exemples d'images erreurs
- Sécurité/qualité : timeouts, "empty outputs", gestion d'images corrompues, fallback CPU/skip

Exemple de pipeline générique (PyTorch + OpenCV)

```python
import cv2, torch
from torchvision import transforms

# 1) Load (attention: OpenCV = BGR)
bgr = cv2.imread("img.jpg", cv2.IMREAD_COLOR)
rgb = cv2.cvtColor(bgr, cv2.COLOR_BGR2RGB)

# 2) Preprocess (à versionner)
tfm = transforms.Compose([
    transforms.ToTensor(),  # uint8[0..255] -> float[0..1], CHW
    transforms.Normalize(mean=[0.485,0.456,0.406],
                         std =[0.229,0.224,0.225]),
])
x = tfm(rgb).unsqueeze(0).cuda()  # (1,C,H,W)

# 3) Model
with torch.inference_mode(), torch.cuda.amp.autocast():
    out = model(x)  # logits / boxes / masks ...

# 4) Postprocess (pseudo-code)
pred = postprocess(out, orig_shape=rgb.shape[:2], score_thr=0.25)

# 5) Business layer
decision = business_rules(pred)  # alert / extract / count / export
```

- Pattern clé : fonctions pures preprocess(), postprocess(), business_rules() (unit-testable)
- Stocker les conventions : format couleur, resizing, coordonnées, seuils, mapping classes
- Debug rapide : conserver rgb, x stats (min/max), sorties brutes et finales (avant règles métier)



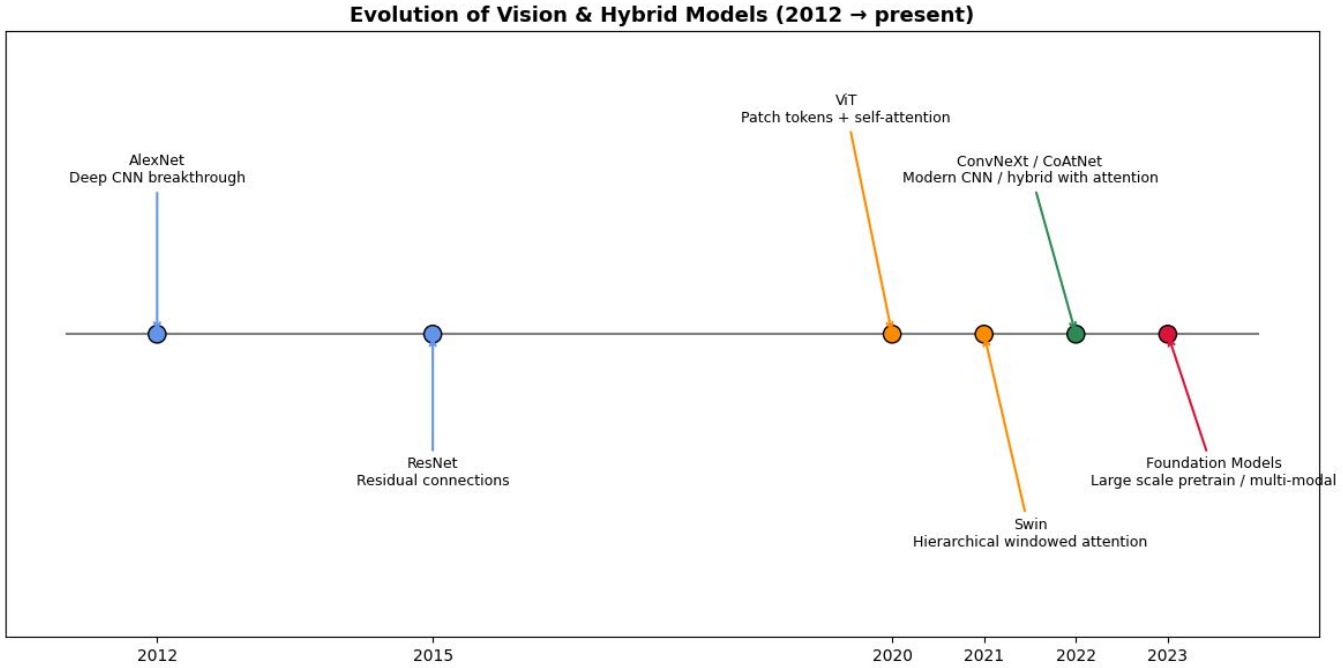
Architectures modernes

Pourquoi les architectures ont (encore) changé

- Les tâches “dense” (détection/segmentation) demandent du **contexte global** + du **multi-scale**
- Les CNN ont un inductive bias (présuppositions utilisée pour prédire des sorties pour de nouvelles entrées) fort (localité, translation)
    - efficace, mais parfois limité en contexte long-range
- Les Transformers ont un biais plus faible
    - scalent mieux avec le **pré-entraînement massif** et le transfert
- En pratique :
    - la victoire ne vient pas d’une “brique magique”, mais de la **scalabilité** + recettes d’entraînement
- Conséquence produit :
    - on réutilise des backbones pré-entraînés “généraux” => adaptation rapide (time-to-market)
- Objectif : comprendre les choix (ViT/Swin/ConvNeXt) en termes de **I/O**, coût, et compatibilité détection/segmentation

Pourquoi les architectures ont (encore) changé

- Les tâche
- Les CNN
  translatio
  - ef
- Les Trans
  - sc
- En pratiqu
  - la
- Conséque
  - or
- Objectif :



Transformers pour la vision : rappel ciblé

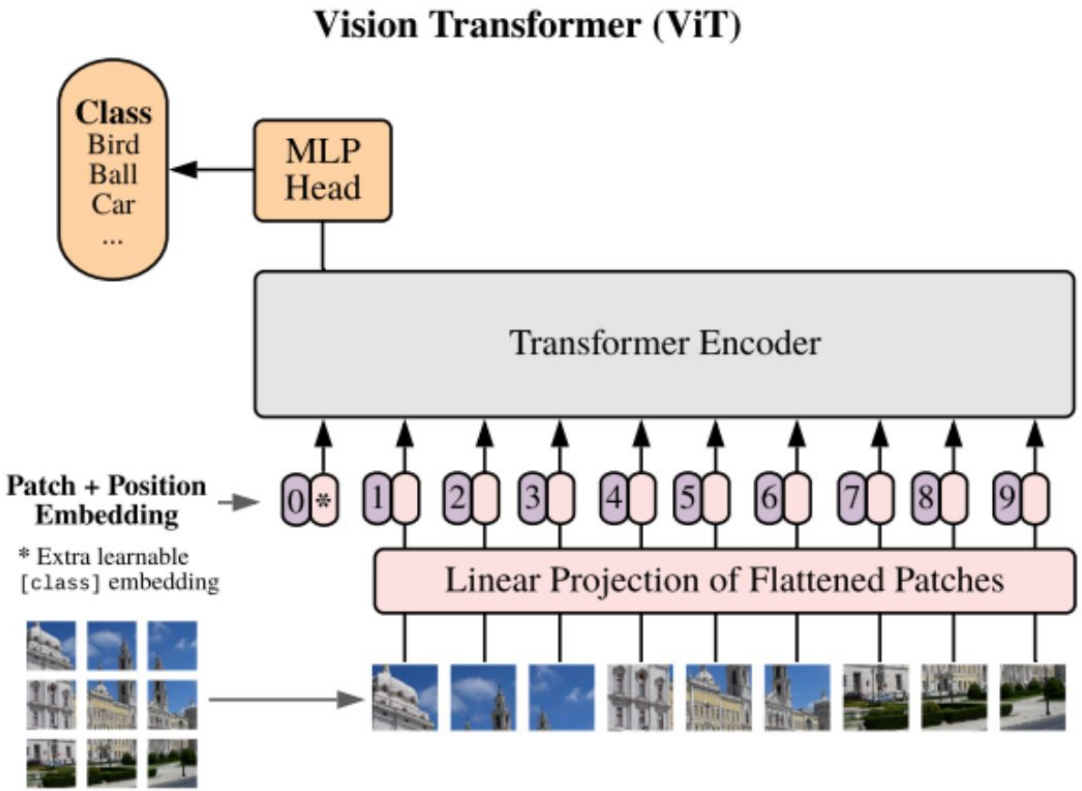
- Entrée = séquence de tokens ($t_1,\dots,t_N$) + embeddings positionnels
- Bloc Transformer
  - **Self-Attention** + MLP + résidus + LayerNorm
- Self-Attention (1 tête) :
  - $\mathrm{Attn}(Q,K,V) = \mathrm{softmax}(QK^T/d)V$
- Intuition vision : chaque patch “regarde” tous les autres => contexte global natif
- Coût clé
  - attention quadratique en N (nombre de tokens) => gros impact sur images HD
- Donc : design moderne = réduire/structurer N (patching, fenêtres, hiérarchie)

ViT : transformer une image en séquence

- Image $x \in R^{H \times W \times C}$, patch size P
- Nombre de tokens : N=H/P * W/P (on découpe l'image en patch)
- Chaque patch est aplati puis projeté :
  - $t_i = W \cdot vec(x_i) + b$ avec $t_i \in R^D$ (transformation linéaire des patchs)
- On ajoute un token spécial [CLS] (souvent) pour classification (comme dans BERT)
- On ajoute un embedding positionnel (appris ou sinusoidal) : $t_i \leftarrow t_i + p_i$
- Sortie classification : head sur [CLS] (on ne garde que [CLS]) ou pooling (mean)
- Point pratique
  - Il faut de gros datasets et un long pré-entraînement pour utiliser ViT

ViT : transformer une image en séquence

- Image $x \in R^{H \times W}$
- Nombre de tol
- Chaque patch
  - $t_i = W \cdot ve$
- On ajoute un t
- On ajoute un e
- Sortie classifi
- Point pratique
  - Il faut d



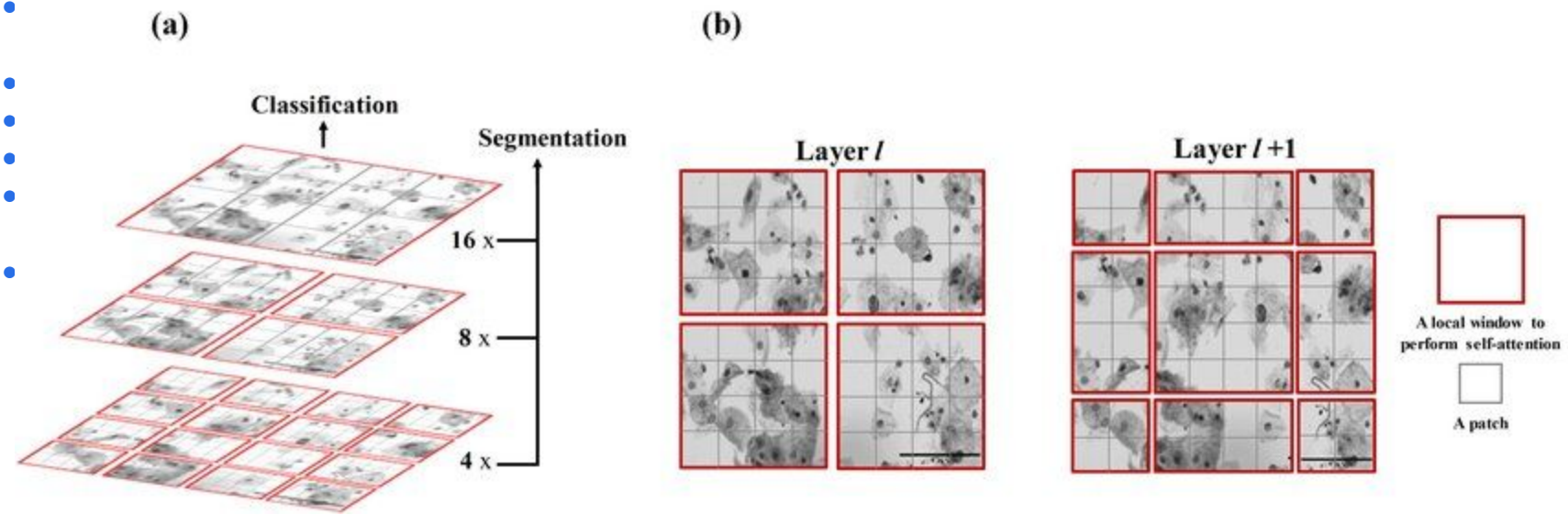
ViT : implications compute/mémoire

- Attention full : complexité $O(N^2 \cdot D)$ (temps) et $O(N^2)$ (mémoire des poids d'attention)
- Exemple mental
  - image 1024×1024, P=16
  - N=4096
  - matrice attention $4096^2 \approx 16{,}7M$ par tête
- Pour du "dense" (détection/segmentation), on veut souvent plus de résolution
  - N explose
- Stratégies
  - patch plus gros (N diminue mais perte détails)
  - attention locale (Swin)
  - pooling/hiérarchie
- En inference : latence dépend fortement de H, W, P pas juste du nombre de paramètres
- Bon réflexe prod : profiler en conditions réelles (résolution, batch, AMP) avant de choisir backbone

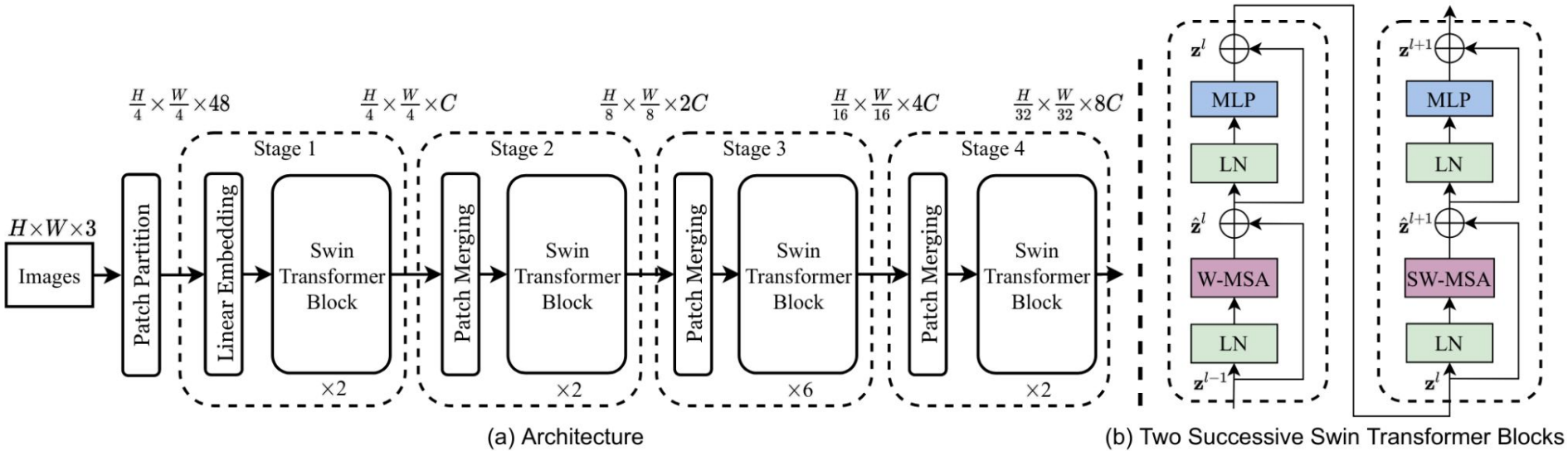
Swin Transformer : rendre le Transformer “multi-scale” et efficace

- Idée : attention **dans des fenêtres** M×M (local attention)
  - => coût ≈ $O(N \cdot M^2)$
- Architecture hiérarchique : on **merge** des patchs (downsampling) => niveaux (comme un CNN)
- Fenêtres “shifted” : on décale les fenêtres entre couches pour faire communiquer les régions
- Résultat : meilleur pour détection/segmentation (besoin multi-scale + détails locaux)
- Lecture système
  - Swin fournit naturellement des feature maps à plusieurs échelles
- Trade-off : localité imposée (bias) mais compute contrôlé → bonne compatibilité prod

Swin Transformer : rendre le Transformer “multi-scale” et efficace



Swin Transformer : rendre le Transformer “multi-scale” et efficace



(a) Architecture

(b) Two Successive Swin Transformer Blocks

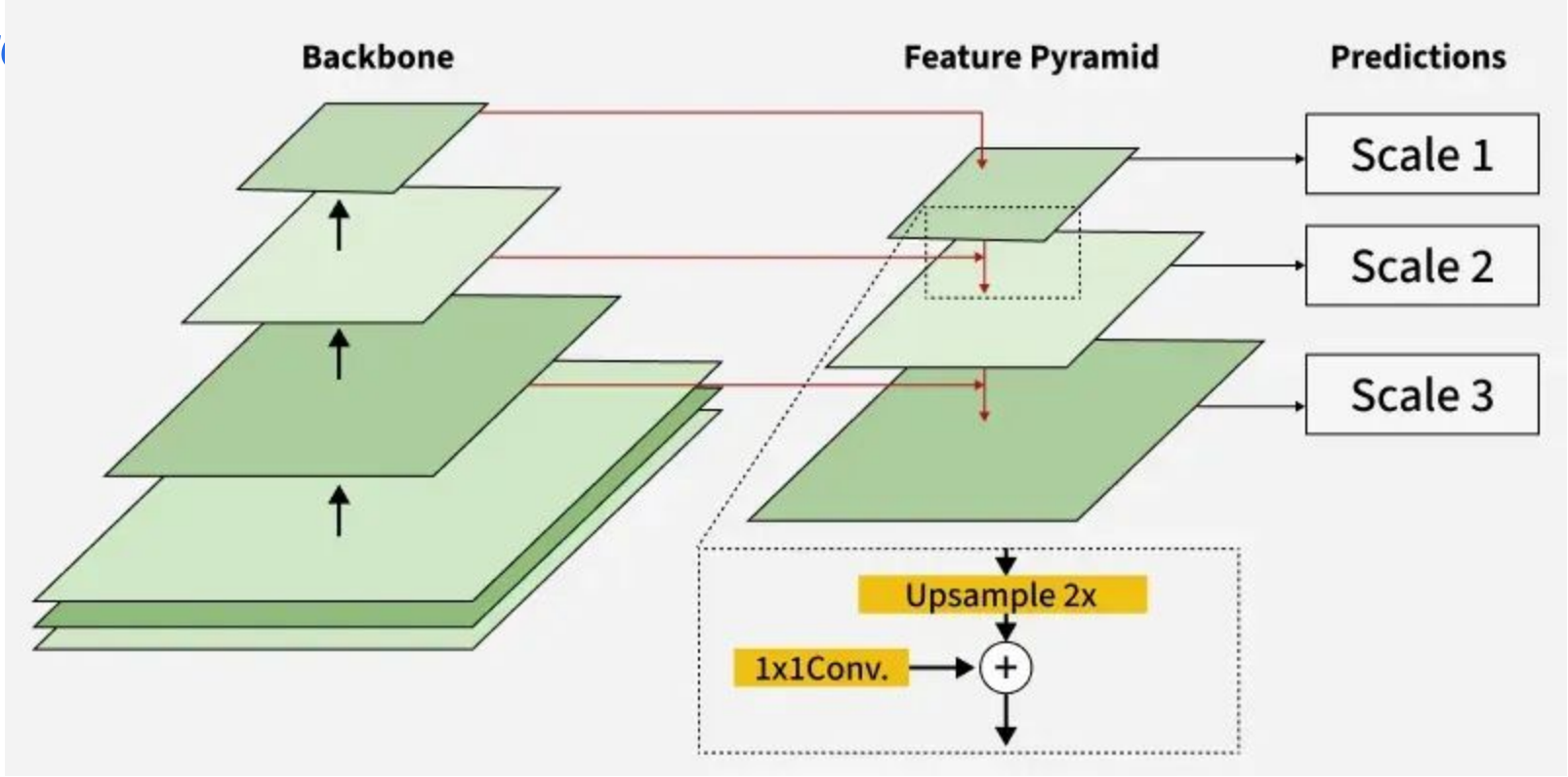
CNN modernes : pourquoi ConvNeXt existe

- Message : “Transformer vs CNN” est trop binaire
  - des CNN modernisés restent très compétitifs
- ConvNeXt (famille) : reprend ResNet et applique des recettes “Transformer-like”
  - normalisation, tailles de kernel, design stage
- Avantage prod fréquent
  - opérations convolutionnelles très optimisées (latence stable, kernels matures)
- Pour certains budgets (edge)
  - CNN bien conçu = meilleur coût/perf que ViT full attention
- Pour dense tasks : CNN + FPN (Feature Pyramid Networks) + head bien réglée = baseline solide et industrialisable
- Conclusion pratique : choisir backbone = contrainte compute/latence + disponibilité des poids + compatibilité head

Feature Pyramid Networks (FPN) : multi-scale “bien fait” pour la détection/segmentation

- Problème motivation
  - objets de tailles très différentes
  - une seule résolution de features ne suffit pas (petits objets vs grands)
- Idée : construire une pyramide de features {P_2, P_3, P_4, P_5} **riches sémantiquement à toutes les échelles**
- Entrée typique : features backbone {C_2, C_3, C_4, C_5} (résolutions décroissantes, sémantique croissante)
- **Top-down + lateral connections** :
  - $P_l = Conv_{3\times3}(Up(P_{l+1}) + Conv_{1\times1}(C_l))$
  - Up = upsampling
- Effet clé
  - on “injecte” la sémantique des couches profondes vers les couches haute résolution
  - meilleur recall sur petits objets
- Utilisation
  - heads de détection/segmentation consomment P_l (anchors/queries/points) selon la taille attendue des objets
- Lecture ingénierie
  - FPN = coût modéré (upsample + conv), gain robuste ; devient souvent une brique standard (Faster/Mask R-CNN, RetinaNet…)

Feature Pyramid Networks (FPN) : multi-scale “bien fait” pour la dét



Pré-entraînement : le “vrai moteur” de la généralisation

- Paradigme : apprendre une représentation f_θ sur un grand corpus puis transférer sur tâche cible
- Supervised large-scale : fonctionne, mais dépend de labels massifs (coût + biais)
- Self-supervised vision : apprentissage sans labels (contraste, reconstruction) → très bon transfert
- Exemples conceptuels :
  - reconstruction (type masked modeling) : prédire des patches masqués
  - contraste : rapprocher vues augmentées d’une même image, éloigner les autres
- Impact entreprise : moins de data labellisée requise pour démarrer un POC robuste
- À retenir : “quel prétrain ?” est souvent plus important que “quel backbone exact ?”

Fine-tuning : stratégies concrètes (et quand les utiliser)

- **Linear probe** : geler backbone, entraîner seulement la head
  - rapide, baseline, peu de risque
- **Full fine-tune** : tout entraîner
  - meilleur perf, mais plus de compute et plus de risque d'overfit
- **Adapters / LoRA** : n'entraîner que de petits modules
  - bon compromis, versioning plus simple
- Heuristique :
  - dataset petit → commencer par linear probe / adapters
  - dataset moyen/grand → full FT possible
- "Catastrophic forgetting" : si FT agressif sur petite data, on peut dégrader la généralité
- Prod : adapters facilitent A/B tests et rollback (on change de "delta" plutôt que tout le modèle)

Exemple code : charger un backbone ViT et extraire des features (générique)

```python
import torch
from transformers import AutoImageProcessor, AutoModel

name = "google/vit-base-patch16-224"  # exemple
proc  = AutoImageProcessor.from_pretrained(name)
vit   = AutoModel.from_pretrained(name).cuda().eval()

inputs = proc(images=pil_image, return_tensors="pt").to("cuda")

with torch.inference_mode(), torch.cuda.amp.autocast():
    out = vit(**inputs)  # out.last_hidden_state: (B, N+1, D)

cls = out.last_hidden_state[:, 0]      # token [CLS]
feat = torch.nn.functional.normalize(cls, dim=-1)  # embedding utilisable pour retrieval
```

- Pattern : backbone pré-entraîné → embedding D → head (classif/det/seg) ou retrieval
- Important : garder trace du preprocessing associé au checkpoint (sinon “silent failure”)

Comment choisir (ViT vs Swin vs CNN moderne) en contexte apprentissage/entreprise

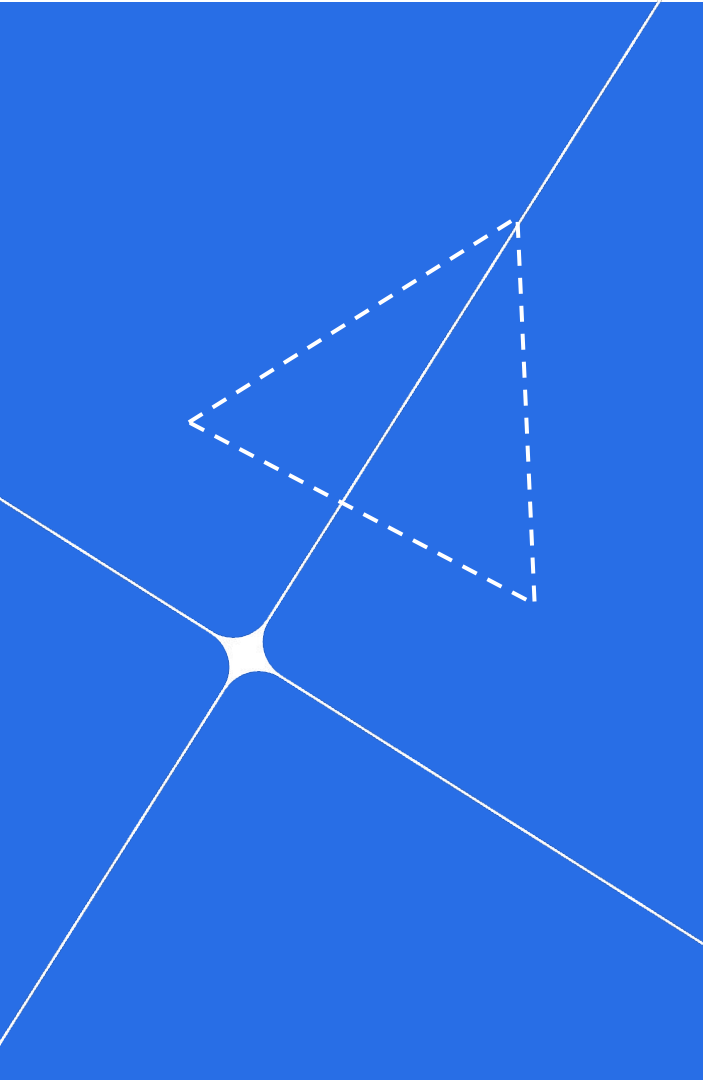
- Si **détection/segmentation** + résolution variable : Swin (ou backbones hiérarchiques) souvent plus naturel
- Si **classification/retrieval** avec gros prétrain dispo : ViT simple et très performant
- Si **edge/latence stricte** : CNN moderne (ou ViT compact) + quantization/AMP (auto mix precision) → plus prévisible
- Si équipe produit : privilégier écosystème, checkpoints, tooling (export ONNX, compat TensorRT)
- Démarrer “pragmatique” : baseline forte (modèle populaire) + protocole d’éval + profiling
- Ensuite seulement : itérer architecture (le gain vient souvent de data + seuils + slices)

Comment choisir (ViT vs Swin vs CNN moderne) en contexte apprentissage/entreprise

Critières (prod)	ViT	Swin	CNN moderne
Latence stable / edge	~	+	++
Haute résolution (HD) sans exploser	-	++	+
Multi-scale natif pour det/seg	~	++	++
Classification / retrieval	++	+	+
Détection temps réel “simple”	~	+	+
Segmentation dense (qualité contours)	~	++	++
Transfer learning (poids disponibles)	++	++	++
Tooling prod (ONNX/TensorRT, kernels matures)	~	+	++
Complexité d’intégration (pré/post)	+	+	++

Transition vers la suite (détection/segmentation modernes)

- Backbones = représentations ; “le vrai travail” dense est dans la **head** + losses + matching + postprocess
- Deux familles à comparer ensuite :
  - Détecteurs “prod” rapides : YOLO-like (one-stage)
  - Détecteurs end-to-end : DETR-like (set prediction)
- Et segmentation moderne : du mask supervisé au **promptable** (SAM)
- Prochain objectif : relier architecture → tâche → métrique → pipeline, sans perdre le fil “produit”



Détection & segmentation modernes : YOLO, DETR, SAM

Pourquoi la détection/segmentation est le cœur de beaucoup de produits CV

- Beaucoup de produits = localiser “où” (objets) et/ou “quelle forme” (masques), pas juste classer
- Cas typiques : inspection (défaut local), retail (comptage), mobilité (piétons/voies), doc AI (zones)
- Contraintes prod : latence (fps), recall sur petits objets, robustesse (blur, occlusions), stabilité post-traitement
- Pipeline standard
  - backbone (multi-scale/FPN-like) → head (boxes/masks) → postprocess (NMS / seuils) → décision
- Deux familles à connaître : **YOLO-like** (dense, NMS) vs **DETR-like** (set prediction, end-to-end)
- Et une 3e voie récente : segmentation **promptable** (SAM) pour annotation/POC rapides et workflows hybrides
- Objectif : comprendre les abstractions I/O + les “points de coût” (compute, postprocess, labels)

YOLO : pourquoi c'est (encore) la baseline industrielle

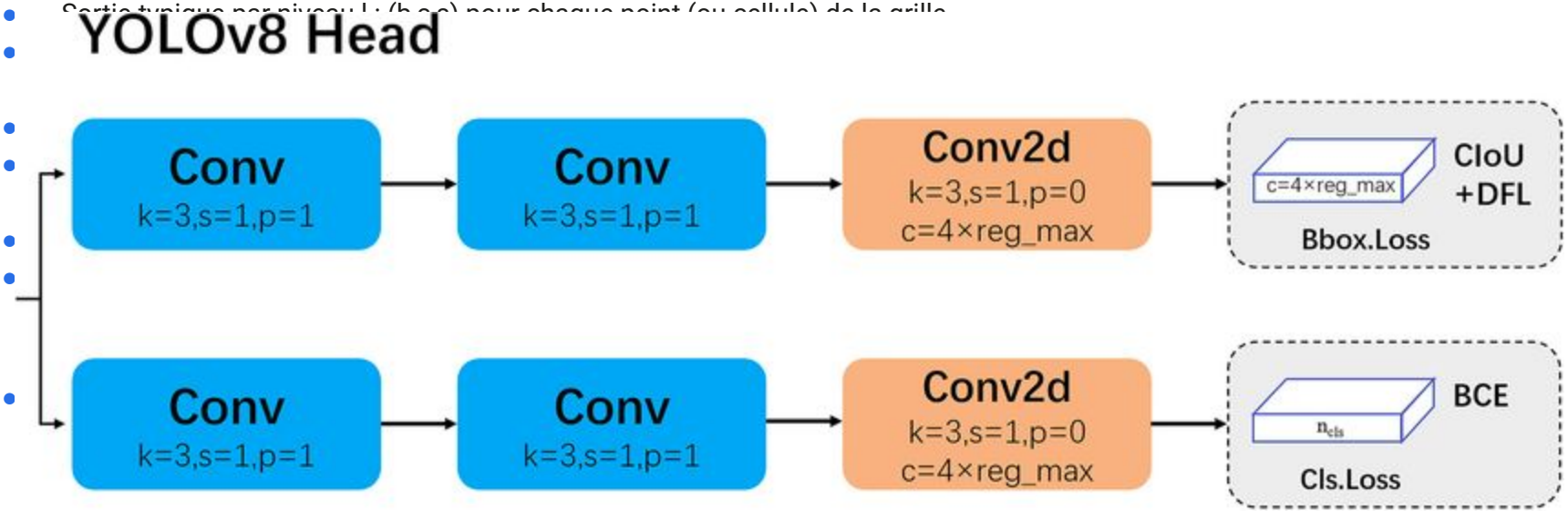
- Design "one-stage"
  - prédire directement des **boîtes + classes** sur une grille multi-échelle (dense prediction)
  - basée sur CNN
- Très bon compromis vitesse/précision
  - adoption massive en POC et intégration produit
- Écosystème pragmatique
  - modèles pré-entraînés, APIs simples, export (ONNX/TensorRT selon stack)
- Tendance récente
  - heads **anchor-free / décorrélées** (classification vs régression) pour simplifier et gagner en robustesse
- YOLOv7 (2022) met l'accent sur l'amélioration de l'entraînement + performance temps réel
- Ultralytics YOLOv8 (2023) popularise une head anchor-free "split" ; YOLO11 (2024) poursuit l'industrialisation (doc + tooling)
- Point d'attention produit
  - la performance réelle dépend beaucoup de NMS/seuils + calibration + distribution terrain

YOLO : sortie, head “découplée” et pertes

- Sortie typique par niveau l : (b,c,s) pour chaque point (ou cellule) de la grille
- Head découplée : branche **cls** (classes) + branche **box** (régression)
  - convergence plus stable (souvent)
- “Anchor-free” : on prédit offsets/boîtes sans choisir un set d’anchors à la main (moins de tuning)
- Perte globale (simplifiée)
  - $L = \lambda_{cls}L_{cls}+\lambda_{box}L_{box}+\lambda_{obj}L_{obj}$
- L_{box} souvent basée IoU/GIoU/CIoU (régression “géométrique” plus alignée métrique)
- Pour classes rares / déséquilibre
  - focal loss devient pertinente, surtout en détection dense
  - focal loss = variante de la cross-entropy qui se concentre sur les exemples difficiles
- Debug pratique : inspecter séparément erreurs “cls” vs “loc” (boîtes) vs “obj” (confidences)

YOLO : sortie, head “découplée” et pertes

- Sortie typique par niveau l : (b,c,c) pour chaque point (ou cellule) de la grille

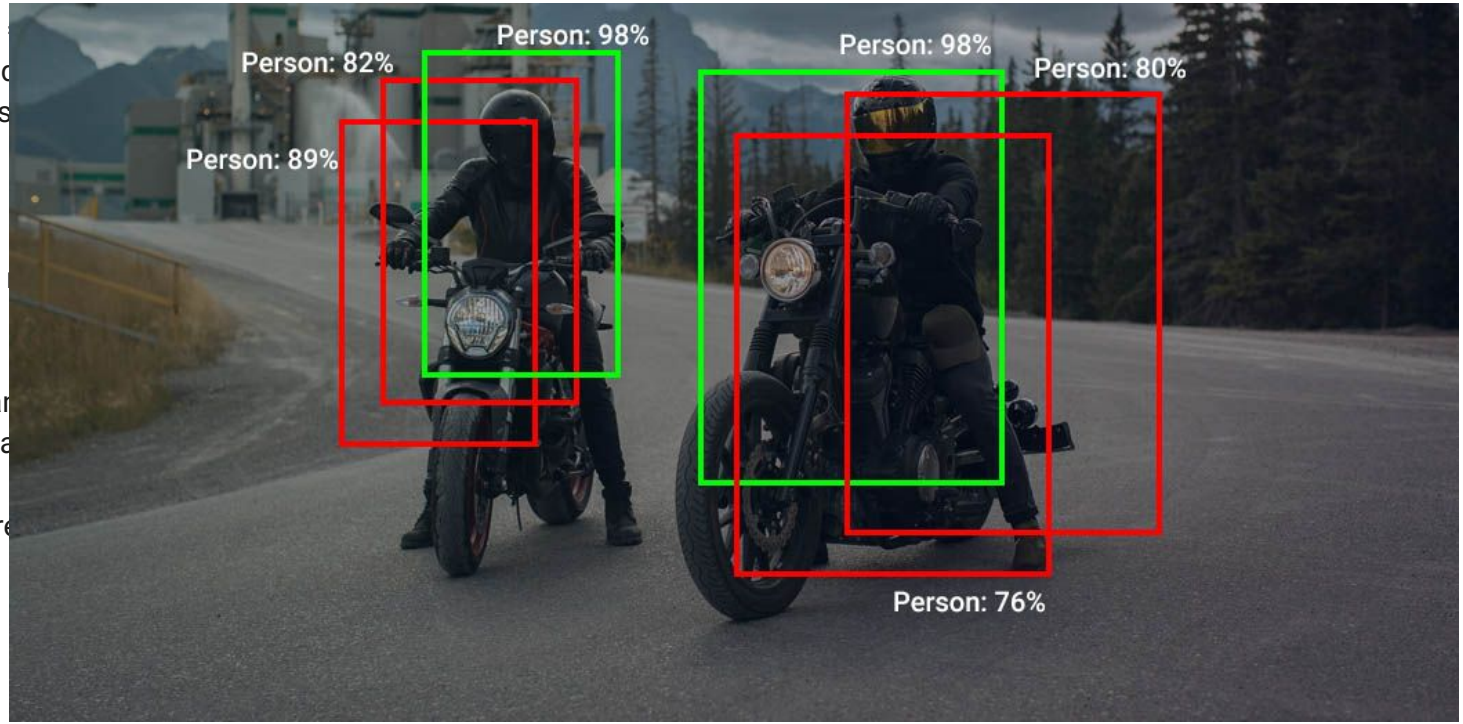


NMS : le post-traitement qui fait (souvent) la différence

- NMS = supprimer doublons de boîtes très proches (même objet) après tri par score
- Pseudo-règle : garder la meilleure boîte, supprimer celles avec IoU > seuil avec une boîte déjà gardée
- Effets prod
  - latence (CPU/GPU)
  - instabilité près des seuils
  - difficulté à rendre le pipeline “end-to-end”
- NMS mal réglé donne
  - trop de doublons (FP)
  - objets ratés (FN) dans scènes denses
- Variantes : Soft-NMS, class-agnostic NMS, batched NMS (accélération)
- Tendance récente :
  - détecteurs “end-to-end” qui **éliminent NMS** (DETR, RT-DETR) ou proposent entraînement NMS-free (YOLOv10)
- Bon réflexe : profiler “model time” vs “postprocess time” (sur vos images, pas sur COCO)

NMS : le post-traitement qui fait (souvent) la différence

- NMS
- Pseud
- Effets
  -
  -
  -
- NMS
  -
  -
- Variar
- Tenda
  -
- Bon re



YOLO en pratique : inference “propre” et intégrable

```python
from ultralytics import YOLO

model = YOLO("yolov8s.pt")  # ou yolo11s.pt selon standard équipe
res = model("img.jpg", imgsz=640, conf=0.25, iou=0.7)  # iou = NMS

boxes = res[0].boxes.xyxy.cpu().numpy()
scores = res[0].boxes.conf.cpu().numpy()
labels = res[0].boxes.cls.cpu().numpy().astype(int)
# => sérialiser en JSON stable, logguer version + params + latence
```

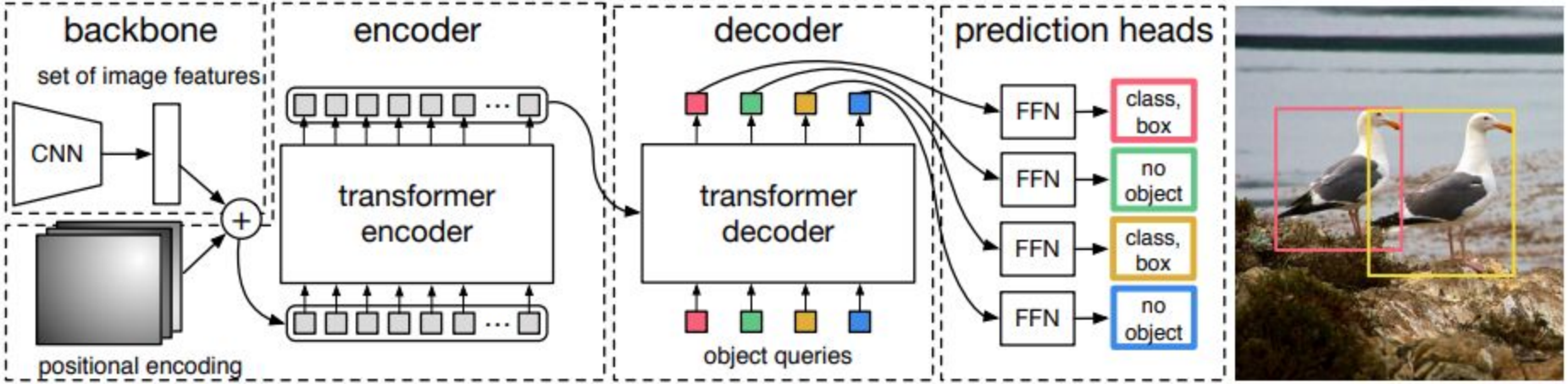
- Paramètres produit : imgsz, conf, iou (NMS), classes autorisées
- Industrialisation : batch inference, AMP, export ONNX/TensorRT, tests de non-régression (slices)
- Observabilité : log “nb boxes”, top scores, taux de vides, latence p50/p95/p99
- Point clé : rendre le postprocess (NMS + mapping coords) déterministe et versionné

DETR : détection comme prédiction d'ensemble (set prediction)

- Motivation : simplifier le pipeline (moins d'heuristiques)
  - pas d'anchors, pas de NMS dans la formulation
- Idée : prédire un **ensemble** de taille fixe N de "slots objets" via des **object queries**
- Architecture : backbone → encoder (context global) → decoder + queries → sorties (b_i,c_i) (boxes + classes)
- Chaque query produit au plus un objet (ou "no-object")
  - on vise des prédictions uniques
- Avantage conceptuel : end-to-end, clair à déboguer (matching explicite, losses structurées)
- Limite historique : entraînement plus lent / plus coûteux sans tricks (d'où Deformable, DINO, RT-DETR…)
- Très utile pour relier Transformer aux contraintes d'un détecteur moderne

DETR : détection comme prédiction d'ensemble (set prediction)

- Motivation : simplifier le pipeline (moins d'heuristiques)



Hungarian matching : la brique math qui remplace NMS/anchors

- Ground truth = ensemble $Y=\{(b_j,c_j)\}_{j=1..M}$, prédictions = $Y'=\{(b'_i,c'_i)\}_{i=1..N}$
- On cherche une permutation σ qui minimise un coût de matching :
  - $\sigma^* = \text{argmin}_\sigma (\text{sum}_{j=1...M} (\text{Coût}((b_j,c_j), (b'_{\sigma(j)}, c'_{\sigma(j)}))))$
- Coût typique
  - classification + distance boîtes (L1) + terme IoU/GIoU (selon papier)
- Perte finale (simplifiée)
  - loss cls + loss box sur les paires matchées + “no-object” sur le reste
- Intuition : chaque objet GT “réserve” une query
  - unicité garantie sans NMS
- Conséquence prod
  - sorties plus stables en scènes denses, mais compute côté encoder/decoder à maîtriser
- Point debug
  - visualiser la matrice de coût (GT × queries) sur quelques images pour comprendre échecs

RT-DETR : rendre DETR réellement temps réel

- Problème : DETR “pur” coûte cher ; RT-DETR vise le temps réel **sans NMS** (end-to-end)
- Idées clés (niveau système)
  - encoder hybride efficace sur features multi-échelle + sélection de queries “de qualité”
- Résultats rapportés (COCO)
  - ex. RT-DETR-R50 ~53% AP (Average precision) et ~108 FPS sur GPU T4 (ordre de grandeur)
- Lecture ingénierie
  - on récupère la stabilité “NMS-free” tout en restant compétitif en latence
- Écosystème
  - implémentations PyTorch open-source
  - variantes (v2/v3) apparaissent
- Quand considérer RT-DETR : scènes denses, besoin de sorties plus stables, volonté d’un pipeline end-to-end
- Attention : toujours profiler avec vos tailles d’images et vos contraintes (CPU postprocess vs GPU compute)

Segmentation supervisée : où elle s'insère (et pourquoi c'est coûteux)

- Sémantique vs instance : en produit, l'instance segmentation sert souvent à **mesurer** et **extraire** (forme)
- Approche classique
  - backbone + FPN + head masques (ex : Mask R-CNN-like) : performant, mais labels coûteux
- Labels masques = coût humain élevé + ambiguïtés (bord, transparence, "partiel" vs "complet")
- Métriques (rappel) : mIoU/Dice ; en instance segmentation : AP masque (COCO mask AP)
- Pattern produit fréquent
  - "détecter d'abord, segmenter ensuite" (boîte → masque) pour réduire compute
- Pour "petites structures"
  - pertes type Dice (ou hybrides) deviennent importantes
- Transition : **foundation models** changent le workflow (moins de training, plus de prompting)

SAM : segmentation promptable

- Idée : un modèle unique qui segmente “ce que vous désignez” via un **prompt** (points, boîte, masque grossier)
- Architecture (haut niveau)
  - image encoder + prompt encoder + mask decoder → masque(s) + score(s)
- SAM (2023) + dataset SA-1B (ordre de grandeur : ~11M images, ~1B masks) : base “foundation”
- Comportement clé
  - **zero-shot** souvent très solide → utile pour POC et pour accélérer l’annotation
- Limites
  - pas “magique” sur domaines très spécifiques (IRMs, micro-déauts) sans adaptation/contrôle qualité
- Pattern entreprise
  - annotation assistée, détourage produit, isolation ROI (region of interest) avant OCR, mesure de surfaces/contours
- Point ingénierie
  - postprocess simple (choix du meilleur masque, nettoyage composantes, export COCO/RLE)

a tree | the fabric | a sheet of corrugated metal | white sack | a yellow shirt | flip-flop | tail light | plaid sarong | the white license plate | a long-sleeved blue and white checkered shirt

plastic bag | pomegranate | a chain
a cardboard sign | the blue basket | plastic basket
the persimmon | a white basket | blue bowl
cardboard box | mango, bread bag, a small glass bowl, ...

white Persian cat | a decorative trim
the blue-green eye | the red velvet chair
the white metal frame
a couch, daybed, a sheep, ...

gravel path | a neatly trimmed bush
the gold finial | a dome-shaped roof
the trellis | the large, yellow estate
the small, white building

blue carpet | black display stand
a vibrant orange 1973 Plymouth Barracuda

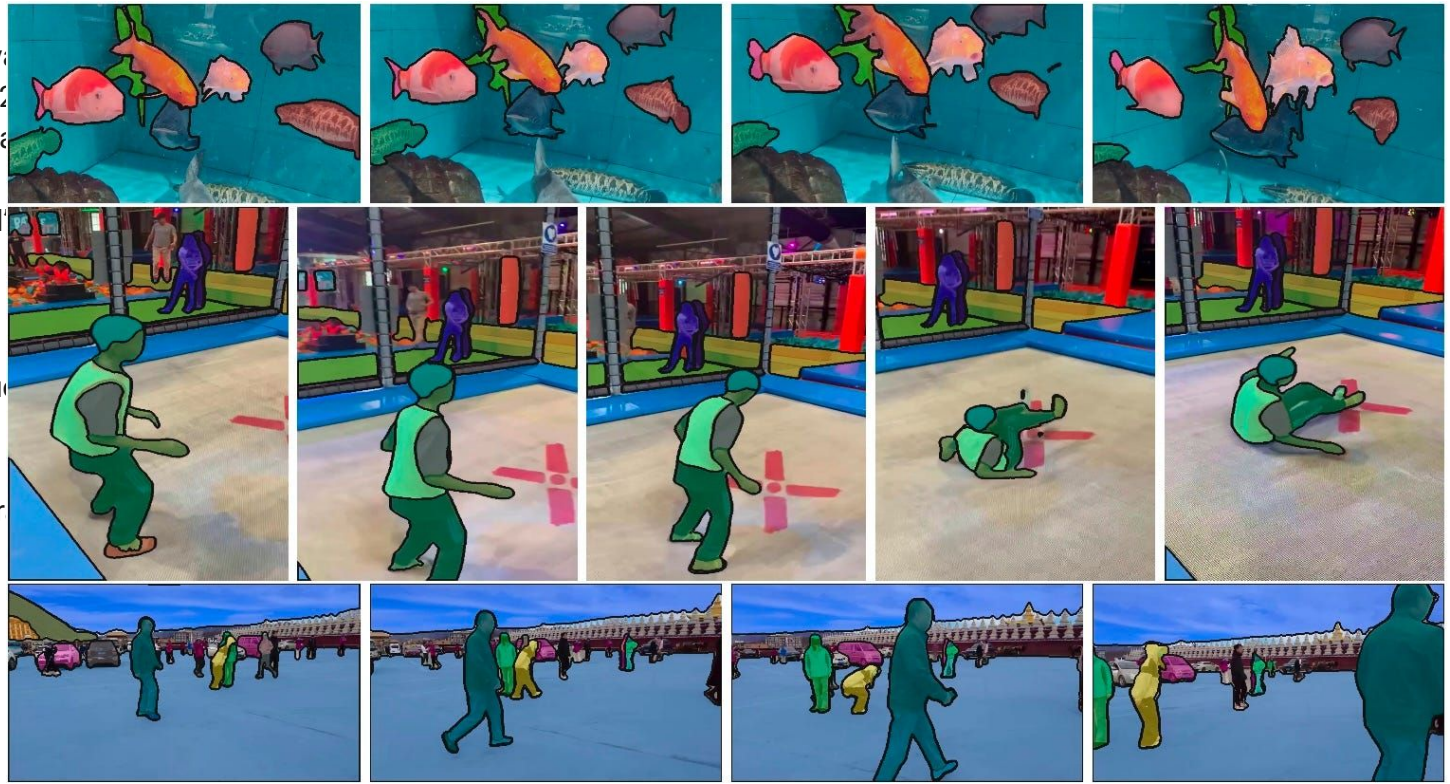
a MacBook | white iPhone
person's left hand

SAM 2 : étendre la segmentation promptable à la vidéo

- Motivation : en vidéo, segmentation = cohérence temporelle + occlusions + contraintes temps réel
- SAM 2 (2024) vise “Segment Anything” **images + vidéos**, avec mémoire “streaming” pour suivi temps réel
- Abstraction produit
  - prompt initial (frame 0) → propagation masques sur frames suivantes (avec corrections)
- Cas d’usage
  - rotoscopie/édition
  - suivi d’objet pour analytics
  - annotation vidéo accélérée
- Risque prod
  - dérive du masque dans le temps (drift)
  - besoin de mécanismes de correction (human-in-the-loop)
- Intégration
  - penser “cache mémoire”, latence frame-to-frame, et stratégie de ré-initialisation

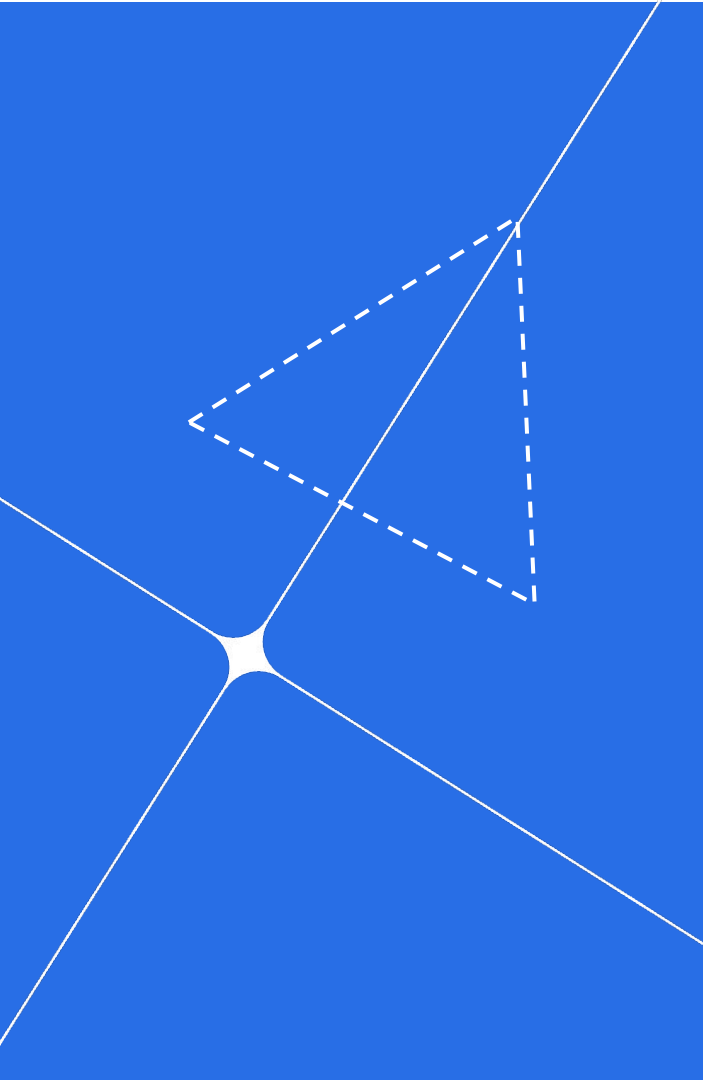
SAM 2 : étendre la segmentation promptable à la vidéo

- Motiva
- SAM 2
- Abstra
  -
- Cas d'
  -
  -
  -
- Risqu
  -
  -
- Intégr
  -



Choisir la bonne famille : règles de décision pragmatiques

- **YOLO** : meilleur “default” pour détection temps réel + intégration simple + écosystème mature
- **DETR/RT-DETR** : intéressant si vous voulez réduire/éliminer NMS, gagner en stabilité sur scènes denses
- **SAM/SAM2** : idéal pour segmentation “à la demande”, accélération annotation, workflows hybrides (detect→segment)
- Pattern très réaliste : YOLO (boîte) → SAM (masque) → mesures/export → dataset → modèle spécialisé si besoin
- En prod
  - profiler *end-to-end* (préprocess + modèle + postprocess) et tester par slices (petits objets, blur, nuit)
- Gouvernance
  - versionner seuils/NMS/prompts et logguer pour pouvoir expliquer une décision
- Règle d’or
  - “baseline forte + protocole d’éval + profiling” avant d’optimiser l’architecture



Ingénierie et industrialisation : faire passer un modèle de vision en production

Pourquoi cette section (et pourquoi maintenant)

- En entreprise, la valeur est souvent dans l'**intégration** : API, latence, robustesse, monitoring, maintenance
- Un modèle SOTA en notebook peut être inutilisable si le **pipeline** est instable (I/O, postprocess, drift)
- Objectif
  - une checklist "ingénieur" pour livrer une brique CV exploitable (POC → MVP → prod)
- Contraintes fréquentes
  - coûts GPU, SLA (Service Level Agreement) latence, volumes, edge/cloud, sécurité, RGPD, reproductibilité
- Approche
  - optimiser **end-to-end** (préprocess + modèle + postprocess + transport + UI)
- Indicateurs de réussite
  - p95/p99 latence, taux d'échecs, stabilité des seuils, taux d'alertes utiles
- Lien direct avec le TP : mêmes patterns (inference, postprocess, export, logs)

Profiling : mesurer avant d'optimiser (sinon on perd du temps)

- Toujours séparer : temps **préprocess**, **inférence GPU**, **postprocess**, **I/O** (lecture/écriture)
- Mesurer p50/p95/p99 (pas juste "moyenne")
  - la prod souffre sur la queue de distribution
- Attention au warm-up GPU (premières itérations demande de charger le modèle) et aux sync implicites (CPU↔GPU)
- Micro-bench. Tester :
  - batch size, résolution, AMP on/off, device (A10/T4/RTX), thread CPU
- Vérifier mémoire
  - VRAM peak, fragmentation, risque OOM (Out Of Memory) sous charge
- Sortie attendue
  - "budget latence" par étape et priorités d'optimisation
- Exemple : parfois NMS ou decoding JPEG coûte plus que le forward pass

Accélérations “sans douleur” : AMP, batching, compilation

- **AMP** (FP16/BF16) : gain débit + baisse VRAM, souvent sans perte ; vérifier mAP/IoU après
- **Batching** : augmente le throughput serveur, mais peut augmenter la latence ; utile si SLA le permet
- torch.inference_mode() + model.eval() : éviter overhead autograd/dropout
- torch.compile (selon versions) : peut accélérer, mais attention compatibilité ops/export
- Pin memory + prefetch data loader (si pipeline vidéo/flux), threads CPU pour décodage
- Optimiser les transferts : garder tensors sur GPU, éviter .cpu() trop tôt
- Toujour: valider qualité après optimisation (régressions subtiles possibles)

Export et runtimes : ONNX / TensorRT / Runtime unifié

- Objectif : exécuter plus vite et plus stable via un moteur d'inférence optimisé (et portable)
- Export ONNX : figer graphe + shapes (ou dynamic shapes) + vérifier opérateurs supportés
- ONNX Runtime : providers CPU/CUDA/TensorRT ; bon pour déploiement cross-platform
- TensorRT : très performant (fusion, kernels, INT8) mais demande calibration/validation
- Points durs : NMS custom, ops non supportées, dynamic shapes complexes (notamment en det/seg)
- Approche pragmatique : exporter d'abord le backbone + head simple, puis intégrer postprocess séparément
- Toujours faire une "parité" : comparer sorties PyTorch vs runtime (tolérance, top-K, IoU)

Quantization : quand et comment (INT8, PTQ vs QAT)

- Pourquoi : réduire taille mémoire et accélérer (surtout edge), parfois nécessaire pour tenir le SLA
- **PTQ** (*Post Training Quantization)*
  - rapide (pas de ré-entraînement), nécessite un set de calibration représentatif
- **QAT** (Quantization aware training)
  - plus long (ré-entraînement), souvent meilleure précision finale, utile si PTQ dégrade trop
- Choisir périmètre : quantifier backbone + head, parfois garder certaines ops en FP16/FP32
- Risques
  - pertes sur petits objets
  - contrastes faibles (détection)
  - instabilités sur certaines couches
- Méthode : mesurer la perte de perf sur vos slices critiques (petits objets, nuit, blur)
- Décision produit : “gain latency” vs “perte recall” (et coût de FP/FN)

Robustesse : le long tail et le “domain shift”

- Drift = changement de distribution
  - nouveau site, nouvelle caméra, saison, packaging, illumination
- La performance globale cache souvent des poches d’échecs
  - *slices* (petits objets, fonds spécifiques)
- Stratégie : définir des “edge cases” et constituer un **jeu de tests vivant** (échantillonnage continu)
- Détection OOD (Out Of Distribution, pragmatique)
  - seuil sur confiance
  - détection d’anomalies sur embeddings
  - règles de sanity
- Human-in-the-loop
  - circuit de revue pour cas incertains + feedback vers data/labels
- Versionner taxonomie + guidelines d’annotation : sinon dérive des labels (label drift)
- En prod, l’objectif est la **stabilité** autant que le score maximal

Monitoring : quoi logger et comment alerter

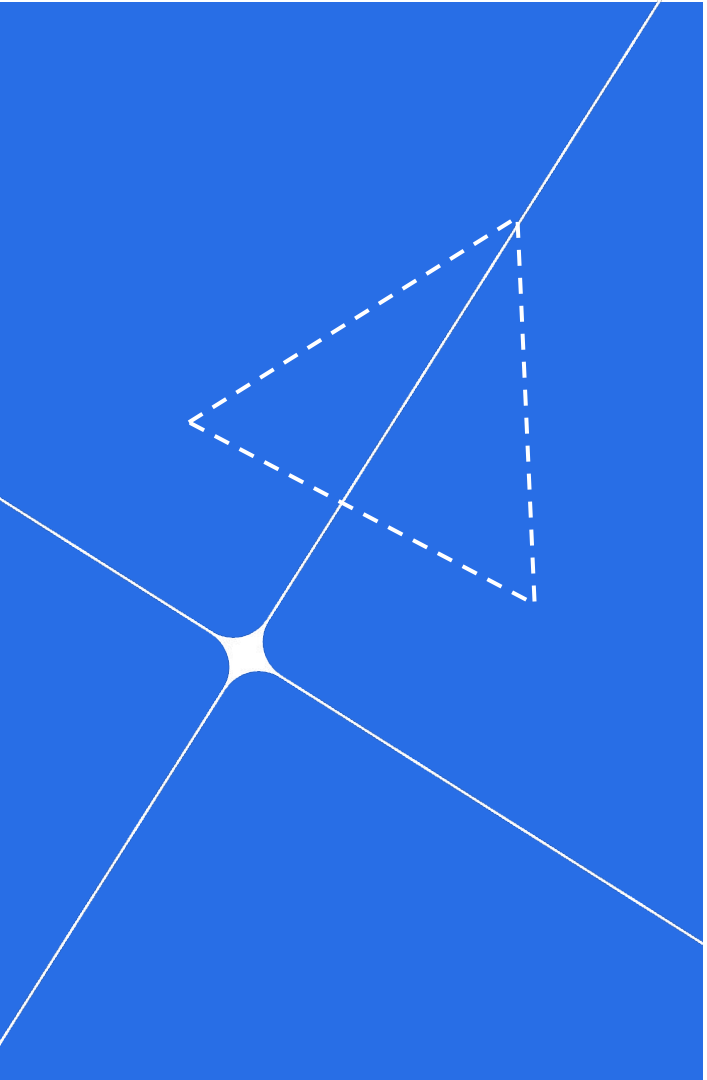
- Mesurer : latence p50/p95/p99, taux d'erreurs (décodage, timeouts), taux de sorties vides
- Mesurer qualité indirecte : distribution des scores, nb objets/image, tailles de boîtes, taux de masques min/max
- Drift monitoring : changements de distribution (histogrammes), embeddings shift, *data quality checks*
- Stocker des exemples : top erreurs / faible confiance / cas OOD (avec respect RGPD)
- Alerting : seuils sur métriques système + dérive statistique, pas juste "accuracy"
- Tests de non-régression : images canoniques + expected outputs (golden set) + tolérances
- Gouvernance : traçabilité version modèle + preprocessing + seuils + config runtime

RGPD / sécurité / PI : points à traiter dès le design

- Données : minimisation (ne pas stocker plus que nécessaire), durée de rétention, anonymisation si possible
- Images de personnes : base légale, information, floutage/masquage, accès restreint, audit
- Modèles et datasets : vérifier licences (poids, datasets, images), droits d'usage en contexte produit
- Sécurité : protection contre inputs malveillants (fichiers corrompus), rate limiting, sandboxing
- Logging : attention aux images sensibles ; privilégier hash + métriques + échantillons contrôlés
- "Explainability" pragmatique : overlays (bbox/masques), scores, règles métier → auditabilité
- Risque réputationnel : biais (démographie, environnement) → évaluer par sous-populations quand pertinent

Conclusion opérationnelle : la checklist “POC → prod”

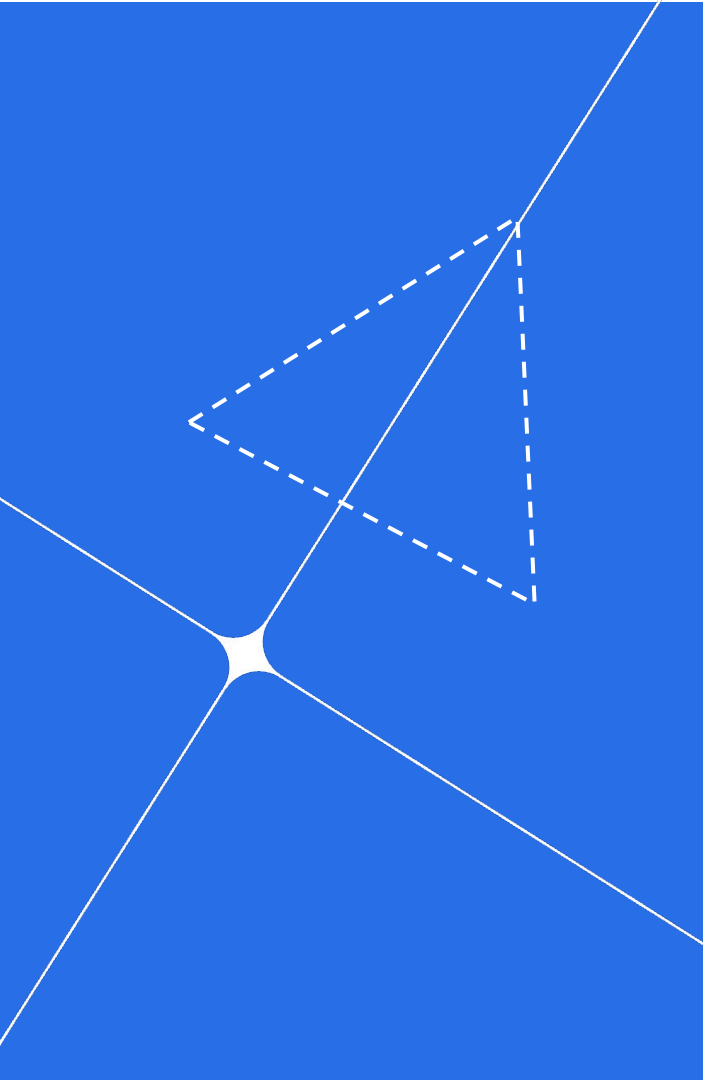
- Définir tâche + métrique + seuils d’acceptation (y compris latence/mémoire)
- Baseline forte (modèle pré-entraîné) + protocole d’éval + slices
- Pipeline déterministe : preprocess/postprocess versionnés, tests unitaires, JSON stable
- Profiling end-to-end, puis optimisations (AMP/batching/export/quantization)
- Observabilité minimale + jeu de tests vivant + boucle de ré-annotation
- Plan de déploiement : A/B test, rollback, gestion versions, documentation
- Résultat : un système CV maintenable, pas une démo fragile



Conclusion

Conclusion : une “recette” CV moderne

- Toujours commencer par **tâche** → **métrique** → **coût d’erreur** (FP/FN) : c’est le contrat produit
- Choisir une **baseline forte** (pré-entraînée) avant d’innover : ViT/Swin/CNN moderne selon contraintes
- Pour le dense : penser **multi-scale** (FPN/hiérarchie) + postprocess (NMS ou set prediction)
- Familles à retenir :
  - **YOLO** = détection temps réel pragmatique, écosystème prod
  - **DETR/RT-DETR** = pipeline plus “end-to-end”, stabilité, moins d’heuristiques
  - **SAM** = segmentation promptable, accélère POC/annotation et workflows hybrides
- La performance utile se gagne en prod via : **pipeline déterministe**, profiling end-to-end, observabilité, slices
- Anticiper le réel : **drift**, long tail, qualité labels, test set vivant, boucle de ré-annotation
- Industrialiser “propre” : versionner modèle + preprocessing + seuils + exports, et intégrer RGPD/licences



En route vers le TP