# Ko mēs vispār cenšamies panākt?

1. Atpazīt burkānus – AI modelis uz GPU
2. Atpazīt nezāļu centrus – AI modelis uz GPU
3. Atpazīt visus zaļos augus – OpenCV uz CPU paralēli GPU procesam
4. Klasterizēt burkānus un noteikt vidējo vertikālo burkānu augšanas līniju vagā, katrā burkānu rindā (vienā vagā var būt vairākas rindas, tipiski 1-4). Reizēm ir nepieciešams lāzerēt tikai noteikta platuma zonu ap katru burkāna rindu.



1. Prioritizējam lāzerēšanu, jo bieži nezāļu ir vairāk nekā spējam nolāzerēt:
	1. Vispirms nezāļu centri, kā jutīgākās nezāļu vietas
	2. Ja vēl ir laiks, tad atlikušās nezāļu daļas (atrodam ar openCV)

# Datu sagatavošana

1. Ievāc attēlus.
2. Atfiltrē nederīgos (nav augu, izplūduši, pārgaismoti umtl.).
3. Anotē manuāli vai ar pretrained AI (kad jau pirms tam bijis pietiekams datu apjoms, lai modelis spētu +/- sakarīgi atpazīt augus ar minimālu vajadzību pēc koriģēšanas).
4. Resize training set (kopā ar visām anotācijām) uz AI modeļa izmēru, mūsu gadījumā 550x550. Šis būtiski samazināja apmācības laiku, jo train skriptam vairs nav N reizes jāskeilo attēls priekš AI modeļa.
5. Empīriski novērojām, ka, mākslīgi palielinot (uzbriedinot) anotāciju poligonus, konkrētā AI konfigurācija ar max\_size = 550 atgriež daudz izmantojamas maskas (nav tik viļņotas formas ap šaurajām burkānu lapām). Skaidri apzināmies, ka pamatā problēma ir tā jau mazu augu skeilošana uz 550x550, bet pagaidām tā ir vēsturiski palikusi problēma, kuru varētu risināt:
	1. lielāks modeļa izmērs, bet ar vieglāku arhitektūru, lai nezaudētu ātrumu
	2. sliding\_window jeb paralēla lielā attēla gabaliņu laišana caur AI

# Apmācība

1. Lietojam uz to brīdi ātrāko instanču segmentēšanas modeli https://github.com/WisconsinAIVision/yolact\_edge
2. Izmanto iebūvēto train.py skriptu:

*python3 train.py --config=yolact\_edge\_resnet50\_config\_wb\_carrot\_weed\_center\_resized --validation\_epoch=-1*

1. AI trenēšanas konfigurācijas fails pievienots. Tas satur arī citas mūsu testa konfigurācijas, taču šajā gadījumā aktuālās vietas ir:





1. Parasti trenēkam un monitorējam mask loss parametru. Kad sāk svārstītis ap kādu plato vērtību, tad apturam.

# apmācības validācija

1. Apmācības laikā skripts ik pēc noteikta iterāciju skaita saglabā tā saucamos čekpoint svarus
2. Mēs uzrakstījām savu skriptu, kurš iteratīvi iziet cauri katram čekpoint svaru failam un uztaisa eval procesu. Vēsturiski mēs ieviesām savu novērtēšanas metriku, lai izvēlētos labākos čekpoint svarus. Būtiskākā atšķirība no klasika IoU, mēs neuzskatām par kļūdu, ja burkāna maska iekļauj zemi ap burkānu, kaut gan validācijas kopā anotācija neiekļauj zemi. Tā ir nevēlama blakne mūsu makslīgajai anotāciju uzbriedināšanai train kopā, kas false positive zemes pikseļu gadījumā neko ļaunu nenodara.



1. Kad tiek pievienota jauna datu kopa, mēs katru reizi modificējam validācijas kopu, pievienojot klāt 10% no jaunajiem attēliem. Ir nošārēta arī pēdējā validācijas kopa, taču mēs to paši neizmantojām, jo mums radās bažas par anotāciju kvalitāti, kas tiešā veidā ietekmē novērtēšanas rezultātus. Lai dzēstu bažas par validācijas kopas anotāciju kvalitāti, mēs izveidojām pavisam nelielu tā saucamo super validācijas kopu ar pēc iespējas precīzāk anotētiem augiem (arī nošārēta).
2. Uz šo brīdi, mūsuprāt, labākie čekpoint svari pēc mūsu custom metrikas redzami šeit ar dzeltenu



1. Jāpiebilst, ka custom metriku validējam pie parametra condicence\_score=0.01, lai atlasītu svarus ar augstāko Recall vērtību. Tālāk uz lauka empīriski piedzenam burkānu un nezāļu konfidences pēc fakta.
2. Nākamais metrikas veids ir redzams grafikā, kur tika anlizēts modeļa robustums pie dažādiem tipveida Recall sliekšņiem 50%-95% pie dažādā filtrētām conf\_score 1-100%. Attiecīgi, jo lielāks conf\_score, jo:
	1. lielākas vidējais Precision vērtības (tiek atfiltrētas zemas pārliecības atrastās instances)
	2. samazinās instanču skaits, kas atbilst attiecīgajam Recall slieksnim (kā blakus efekts tiek atfiltrētas instances, kuras samazina statistisko rādītāju)
	3. palielinās FN instanču skaits procentos (tas pats blakus efekts kā “b” punktā)



# testa kods

1. Ir pievienots pavisam neliels kodiņš, kuru uzrakstījām, kad sākām pāreju uz Rust. Kods faktiski izsauc Python bāzētu AI inferenci. Koda mērķis bija saprast, vai tehniski tas ir izdarāms un novērtēt vai nav lieku overhead, kad izsauc AI framework no Rust. Kāpēc tāda pieeja? AI framework noteikti šobrīd neprotam pārrakstīt uz Rust un pagaidām nemēģinājām, vai to var konvertēt piemēram uz ONNX un tad pa taisno izsaukt Rust. Faktiski tas tika atstāts kā “saldais ēdiens”, kamēr tiek pārrakstīta visa pārējā loģika no Python uz Rust.
2. Uz mūsu produkcijas dzelža šis pats kods nomērīja vidēji 7.34ms uz RunInference bloku (būtiska piebilde, ko nepieminēju Zoom laikā – tas ir kopējais laiks, ieskaitot dažas papildus funkcijas pirms un pēc faktiskās inferences. Skatīt kodā yolact\_edge\_interface.py) ar konfigurāciju resnet50 TRT FP16.