Issues w/ Deep Learning / CNNs

- overfitting
- vanishing & exploding gradients ⇒ slow learning
- hyper parameter selection
  - optimal kernel sizes
  - # of filters (feature maps) per layer
  - # of layers
  - pooling & stride sizes, type of pooling
  - learning rates (function of the minibatch size)
  - weight decay (amount of l_2 regularization on your weights)

bag of tricks, exploration

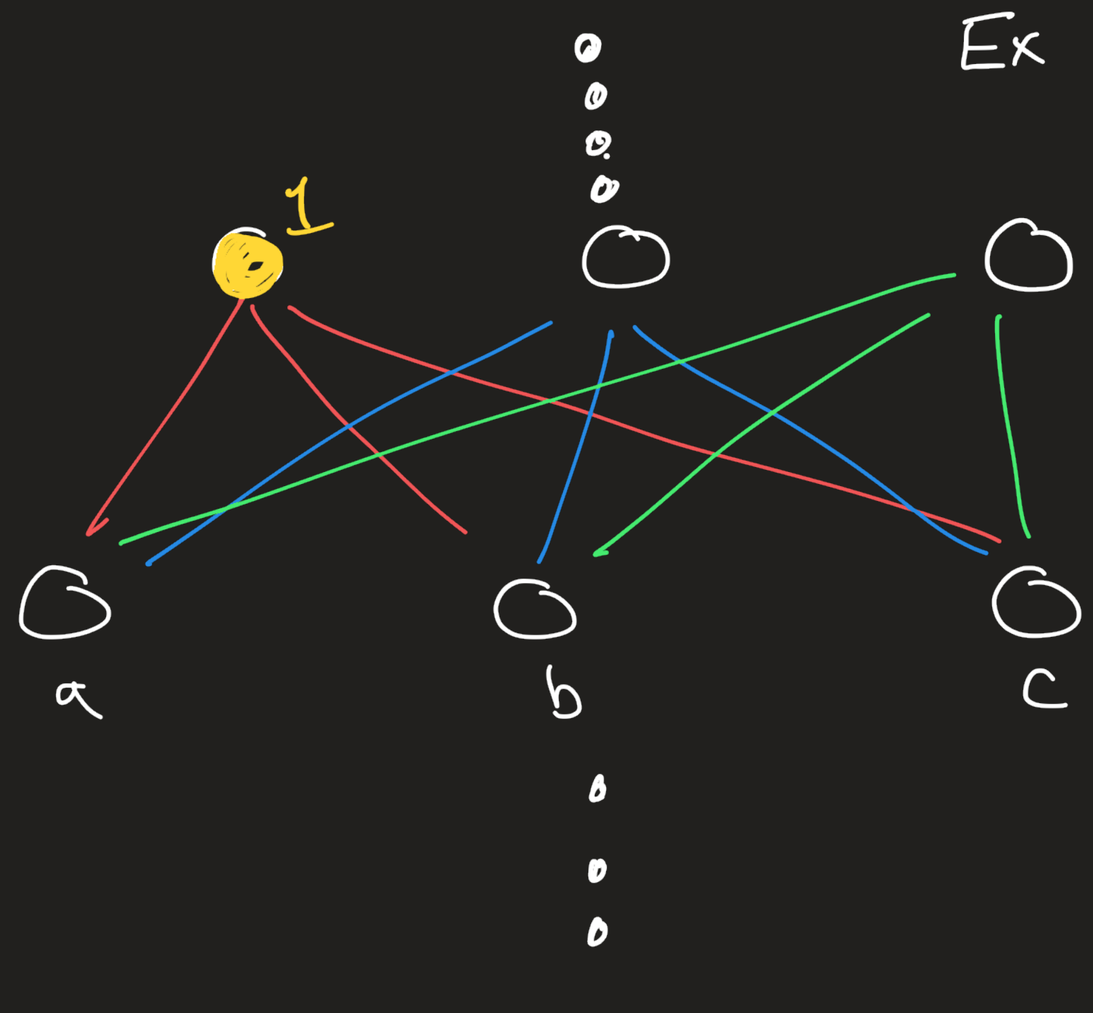
Remedies for overfitting & the vanishing/exploding gradients problem

Dropout (2014?)

- form of regularization introduced to prevent overfitting
- led to a series of follow-ups: DropConnect, etc.

Intuition: to avoid "feature co-adaptation" (where some neurons in one layer learn brittle, non-generalizing combinations of features in the previous layer)

randomly drop some activations from the previous layer to zero, so neurons have to learn to compute robust features during training. These random dropouts change at each minibatch during training.



Ex

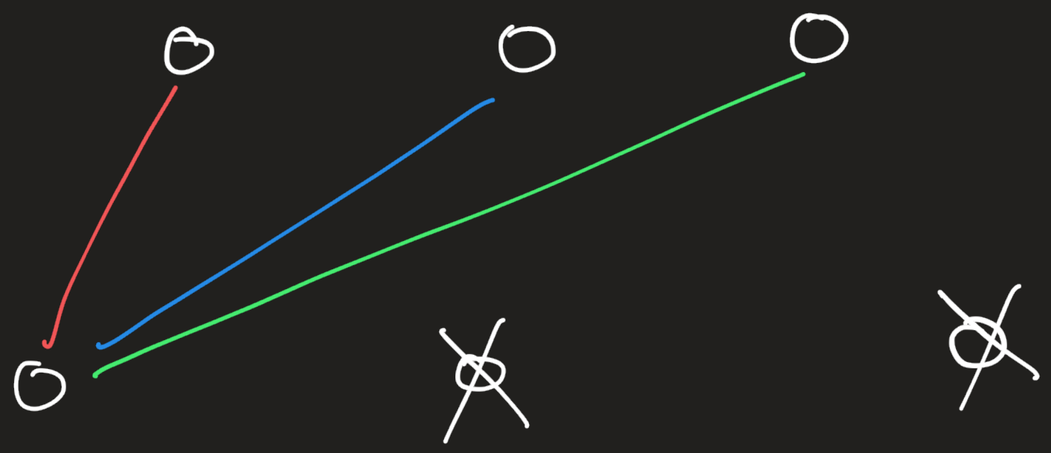
FC

could be that 1 neuron learns a feature

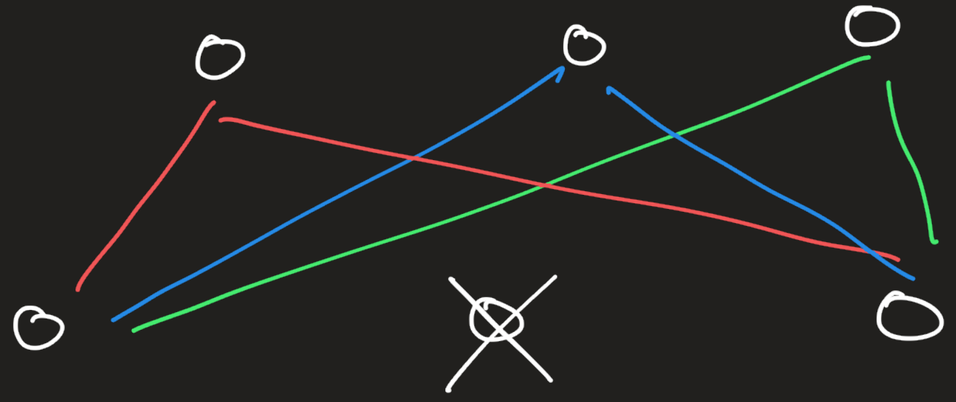
$\sigma(.999a + .0005b + .0005c)$

where <u>any</u> slight change to these weights performs poorly

dropout



minibatch 1



minibatch 2

on every minibatch
just update the weights
connecting the non-dropped neurons

Adding dropout between layers $l-1$ and l of our NN is equivalent to changing our output formulas to:

$$a^l = w^l\left[v^{l-1} \odot o^{l-1}\right] + b^l$$

where $v^{l-1} \in \mathbb{R}^{n_{l-1}}$ is a Bern(p) vector equivalent to zeroing outputs of the neurons where the entries of v^{l-1} are zeros.

And note, v^{l-1} is resampled at each minibatch.

This is equivalent to randomly selecting a subnetwork of our NN and updating its weights via backprop.

How to use a network trained using dropout at test time?

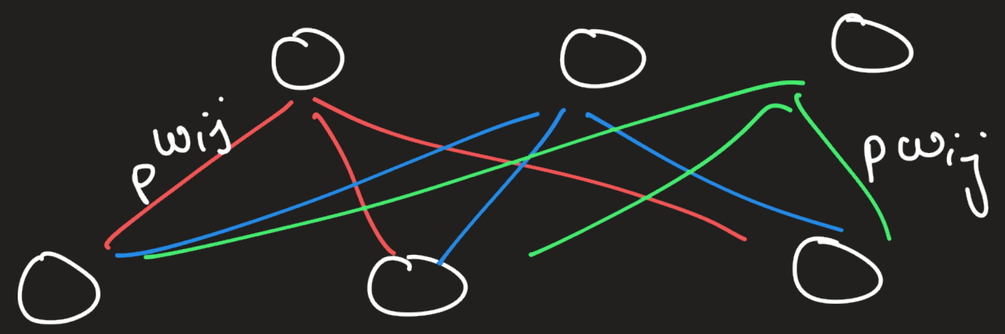
(1) could dropout neurons as before and use the subnetwork to predict

standard

(2) use the average activation

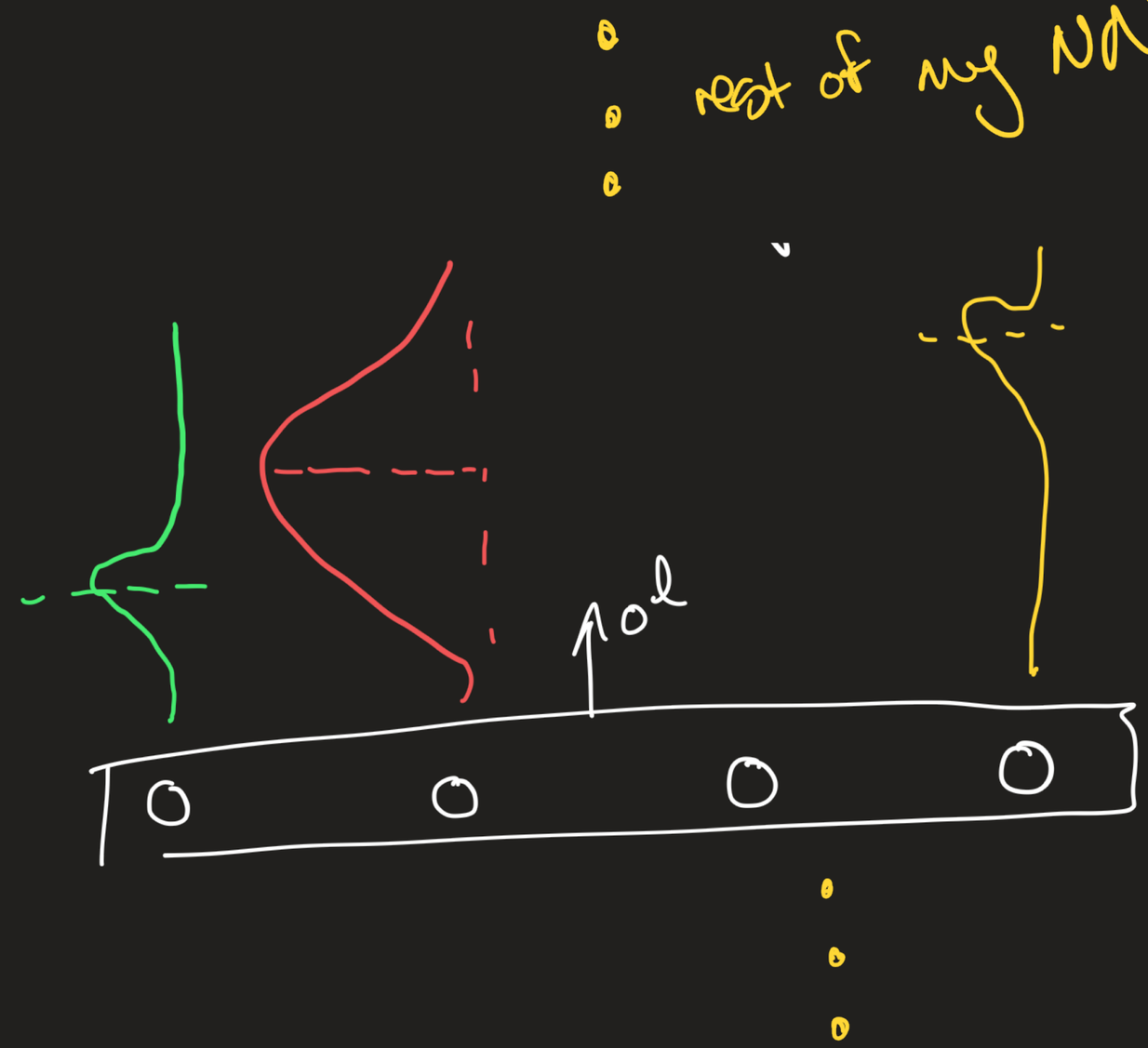
$$\mathbb{E}\, a_l = w^l\left(\mathbb{E}\, a^{l-1}\right) \circ o^{l-1} + b^l$$

$$= p\, w^l o^{l-1} + b^l$$



Batch Normalization (2015)

Idea: the feature distributions at each layer vary wildly, so normalize them so they look like standard gaussians



In convex ML we saw the advantage of normalizing our input data so the features are on the same scale

ex: training to predict credit default probs

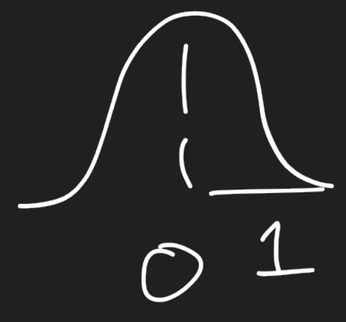
$x = [\text{salary} \quad \text{age}]$

to get the features on the same scale we normalized (say scikit standard scalar)

$$x \mapsto \frac{x - \mathbb{E}x}{\sqrt{\mathrm{var}(x)}}$$

zero-mean & rescale all our features to have std 1 (compute z-scores)

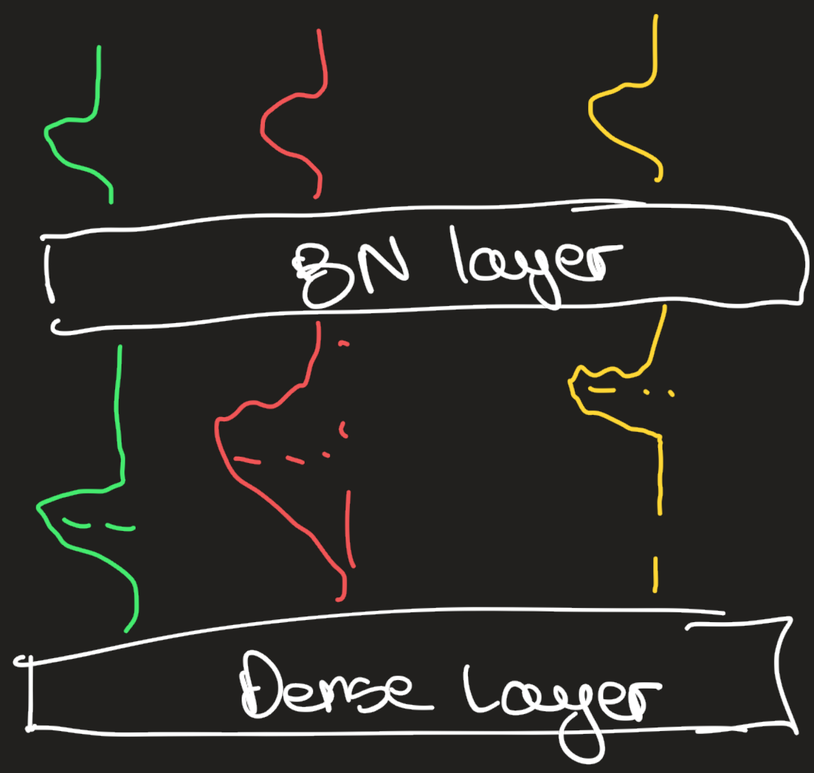
salary z-scores



age z-scores

getting the features on the same scale makes your problem computationally easier (the optim problem is better conditioned)

Idea of BN: add normalization layer after a regular layer to rescale and center the feature distributions, to make the learning problem easier



Batch normalization:

Between layer l and $l-1$ add a BN layer:

$$a^l = w^l BN_{\gamma^l, \beta^l}(o^{l-1}) + b^l$$

$$BN_{\gamma^l, \beta^l}(o^{l-1}) = \gamma^l \frac{o^{l-1} - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}} + \beta^l$$

γ^l scale param $\in \mathbb{R}^{l-1}$

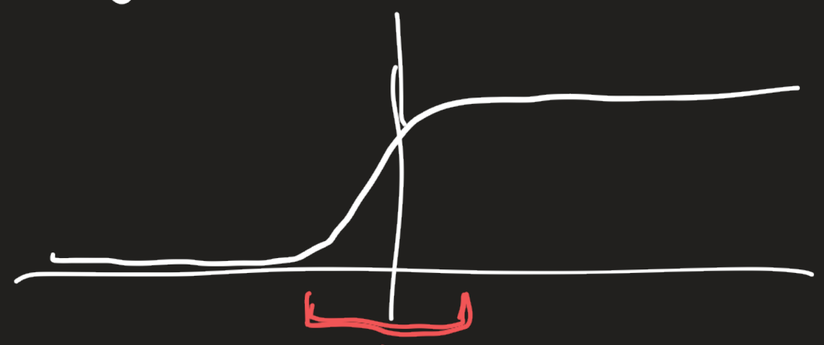
β^l shift parameter $\in \mathbb{R}^{l-1}$

are learned

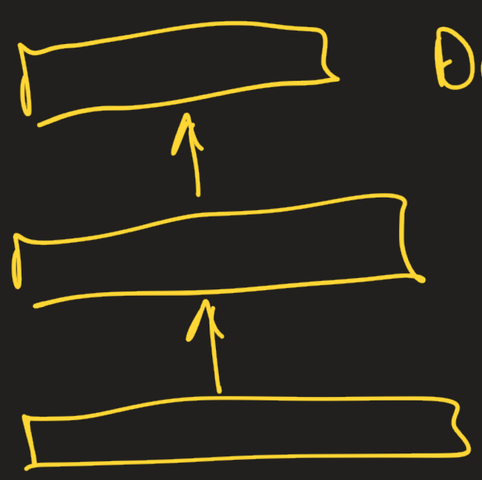
where μ_B = minibatch mean $= \frac{1}{m} \sum_{i=1}^{m} (o^{l-1})_i$

σ_B^2 = minibatch variance $= \frac{1}{m} \sum_{i=1}^{m} ((o^{l-1})_i - \mu_B)^2$

Why do we use γ and β?



learning γ and β help to ensure we aren't limited to the linear portion of our activation function



How do we use batch normalization layers at inference time?

1) After fitting our NN, run all our training data through it and for each layer compute

$$\mu^{\ell} = \frac{1}{n_{train}} \sum_{i=1}^{n_{train}} (O^{\ell-1})_i \quad \in \mathbb{R}^{n_{\ell-1}}$$

$$\sigma^{\ell} = \frac{1}{n_{train}} \sum_{i=1}^{n_{train}} ((O^{\ell-1})_i - \mu^{\ell})^2 \quad \in \mathbb{R}^{n_{\ell-1}}$$

and use these in our batch normalization layers

2) or maintain moving average estimates of the population mean & variances

$$\mu^{\ell} \leftarrow \beta_1 \mu^{\ell} + (1-\beta_1)\mu_B^{\ell} \quad \text{for each minibatch}$$

$$\sigma^{\ell} \leftarrow \beta_2 \sigma^{\ell} + (1-\beta_2)\sigma_B^{\ell} \quad \text{for each minibatch}$$

and use μ^{ℓ} and σ^{ℓ} at inference time.

Data augmentation (addressing overfitting)

Idea: