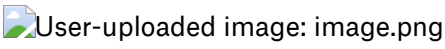
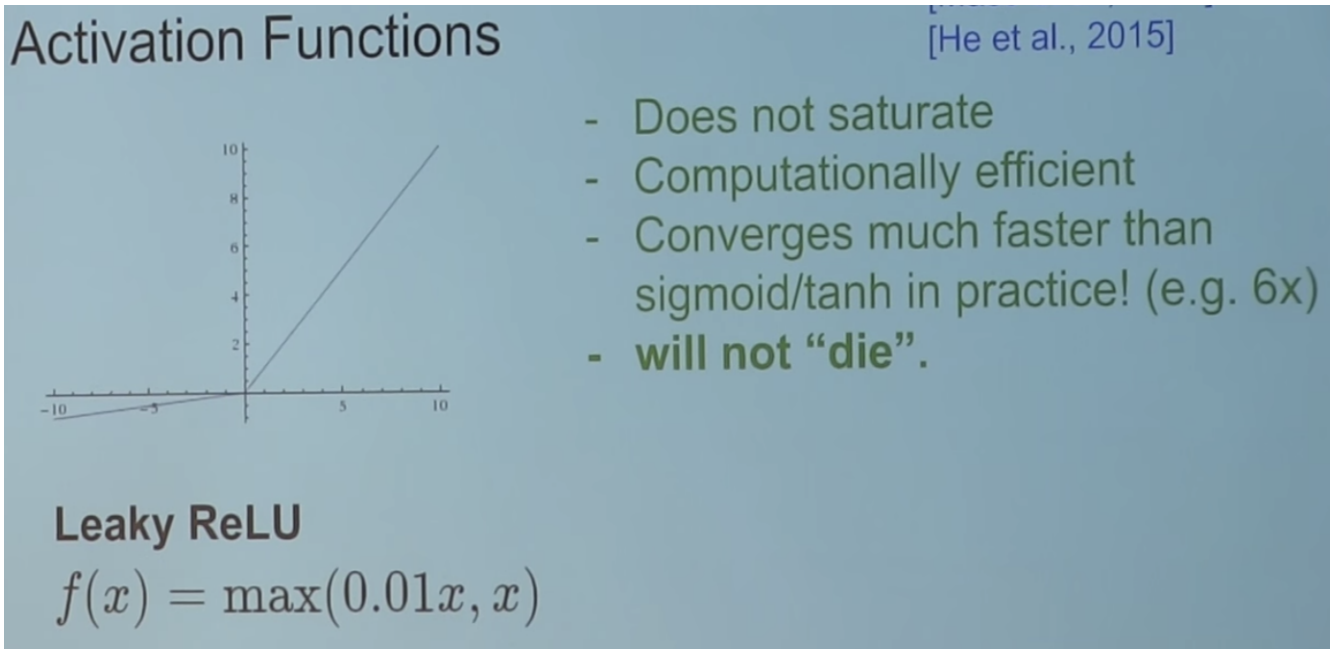
neural nets basics

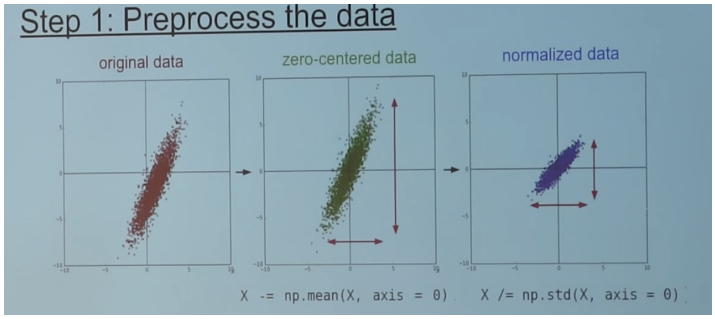
- you can think of
  - add operations as gradient distributors during backprop
  - max operators (in max pooling) as gradient routers that route gradient to the max value.
- during forward propagation, if you can store the activations it can help to compute gradients faster but this is a time-memory tradeoff → see gradient checkpointing work by openai
- when using softmax in the last layer of a neural net, that is basically a linear separator over the features computed by the previous layer. So you can think of neural nets as warping space such that the last layer outputs features in a space such that different categories are linearly separable.
- people rarely ever train CNNs from scratch. use a model zoo to grab some trained weights. cuz if you are performing a vision task, that pre training almost definitely helps.
- because of pre trained weights you can also get pretty good performance with a small dataset size, vision has really made advances in transfer learning cuz its easier in vision → take advantage of pretrained weights and don't be afraid to work with small datasets.
- Rumelhart et al. were the first to popularize backprop around 1986 (not invented then)
- in 2006, Hinton, Ruslan et al trained a network layer by layer with an unsupervised objective, used this as initialization and then stacked them and trained them end to end → first successes of neural nets. nowadays we don't need to do this.
- 2010 Microsoft research had a breakthru in speech recognition where they replaced some components of speech pipeline with neural net backed components. and 2012 was AlexNet that showed considerable improvement in error rate on ImageNet.
- the advantage of $tanh$ over sigmoid is that it is 0 centered, so not all gradients for all weights of a unit need to have the same sign → this might be why we see faster convergence in these

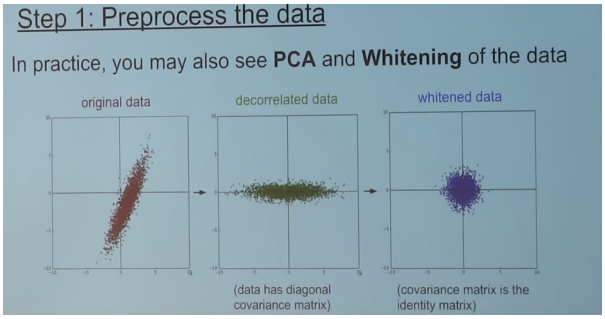


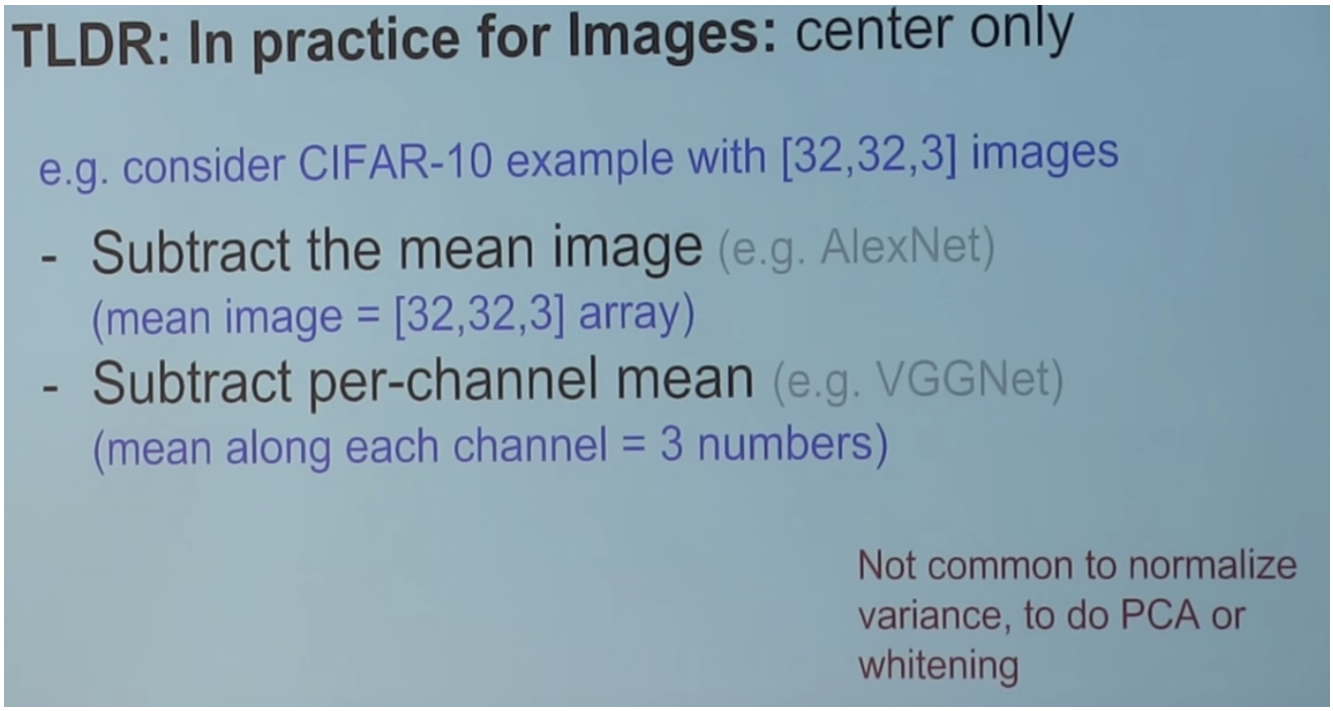
- But $tanh$ still gets saturated with very big or very small numbers → so enter ReLU, but need to initialize so that we get a good number of non-zero gradients at first. pick a biggish positive bias to enable this when you initialize.
- Leaky ReLU is another way to get around this



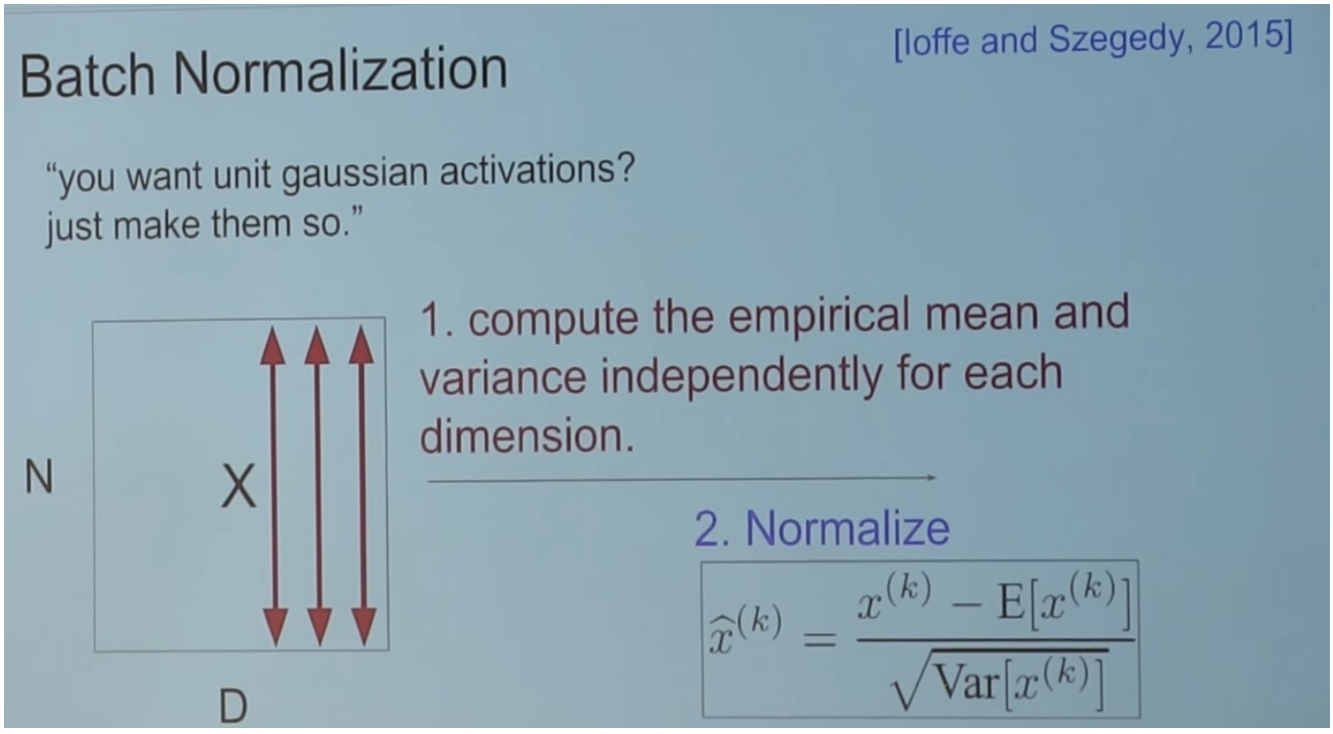
- Data preprocessing



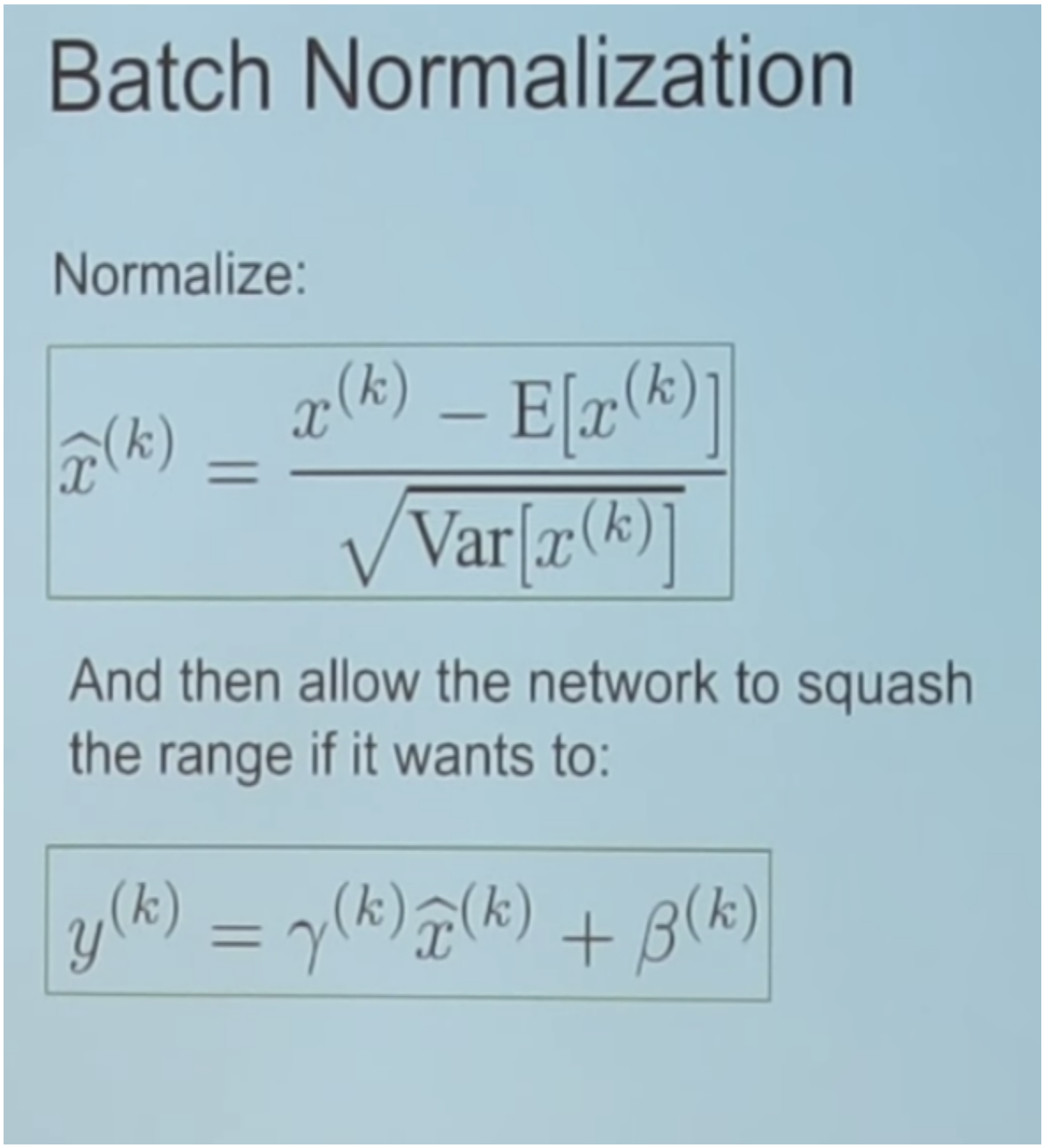




- Initialize weights to small random numbers. Initializing all weights to the same value results in symmetry and all gradients are same. You should break this symmetry or it'll be a disaster.
- In a very deep networks (approx 5/10+ layers), we don't want to initialize fully randomly with small numbers. if we did this as the numbers progressed, they would all get multiplied with tiny numbers, become tiny, become even tinier and activations would be become really really close to 0. This isn't good for learning → gradients become very very small too. so instead, use a strategy where the weights are initialized to a 0 mean gaussian with a variance inversely proportional to the number of neurons in the layer. So more the neurons, lesser each weight so that the distribution of outputs from them stays consistent (not too low) so that gradients can propagate back → Xavier initialization.
- Or another way is batch norm:

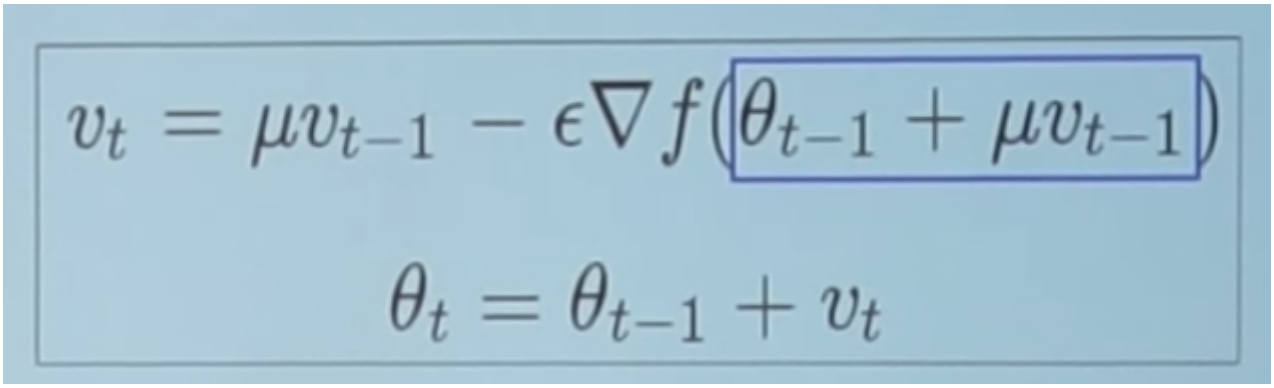


- Batch norm reduces dependence on good initialization and allows higher learning rates. When using Batch norm, we want to allow the network to skew the mean and variance of the activations, if that's where gradient descent pushes it. So we will do



- when you train a neural net with softmax the loss might look like its staying the same, but accuracy might be going up. this is because even if the probability assigned to correct class is a tiny bit more than that assigned to incorrect class, you get correct classification but loss term is still not doing very well. later on in training, you will see cost go down as well as probability gets more skewed.
- the problem with vanilla SGD is that when you get to the bottom of the vessel, it can keep overshooting, and it doesn't build up momentum when moving in the same direction where momentum could help speed of convergence.
- read about momentum and how it improves convergence by moving faster if you keep going in the same direction and becomes like a damped oscillating ball coming to a stop when you are close to the bottom of the vessel → in deep learning book notes

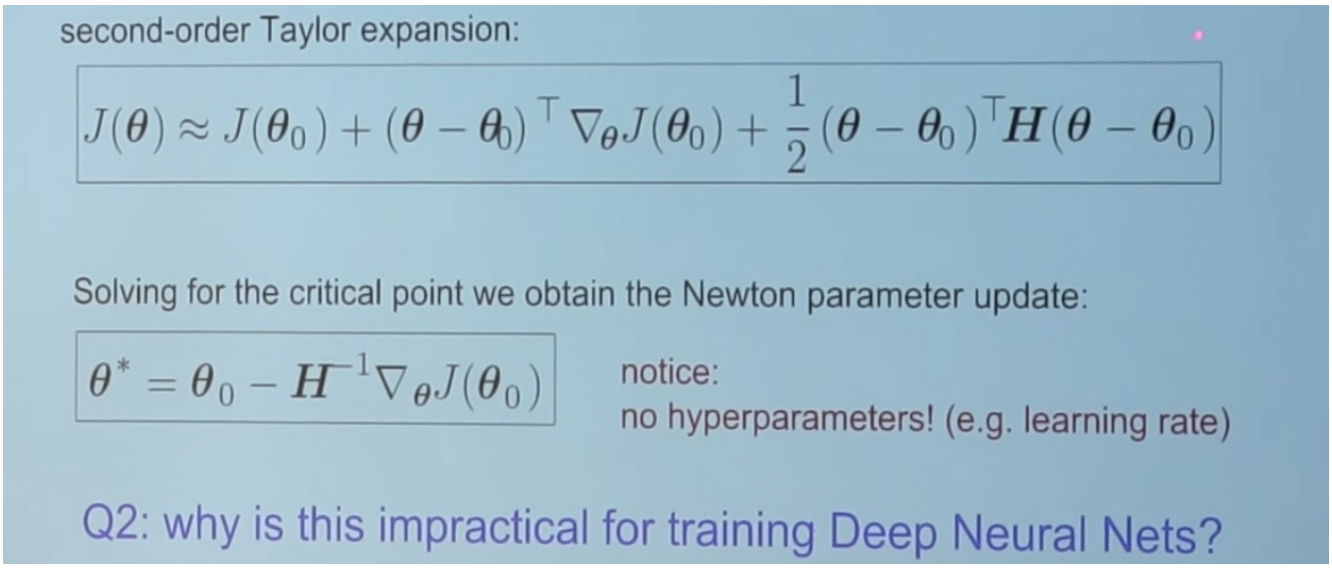
- Nesterov momentum involves momentum term + computing the gradient at next step instead of current step → turns out to work better than vanilla momentum in practice



$$v_t = \mu v_{t-1} - \epsilon \nabla f(\theta_{t-1} + \mu v_{t-1})$$

$$\theta_t = \theta_{t-1} + v_t$$

- Read up on Adagrad, RMSProp and Adam from deep learning book notes
- Use some kind of decay (like exponential decay in your learning rates so that as you converge to the bottom of the vessel, you have low energy and you just settle down instead of bouncing around too much.
- Second order methods for optimization summary → Hessian is too big, there is a way to make it work by iteratively computing Hessian without inverting but still needs a ton of memory, LBFGS does not need as much memory but suffers from randomness, needs whole batch to fit into memory which isn't practical.



$$\theta^* = \theta_0 - H^{-1} \nabla_\theta J(\theta_0)$$

- Quasi-Newton methods (**BGFS** most popular):
  instead of inverting the Hessian (O(n^3)), approximate inverse Hessian with rank 1 updates over time (O(n^2) each).

- **L-BFGS** (Limited memory BFGS):
  Does not form/store the full inverse Hessian.

L-BFGS

- **Usually works very well in full batch, deterministic mode**
  i.e. if you have a single, deterministic f(x) then L-BFGS will probably work very nicely

- **Does not transfer very well to mini-batch setting**. Gives bad results. Adapting L-BFGS to large-scale, stochastic setting is an active area of research.

- **Adam** is a good default choice in most cases

- If you can afford to do full batch updates then try out **L-BFGS** (and don't forget to disable all sources of noise)

- talks about ensembles, computational tradeoff and dropout next → won't go into these here.
- dropping out activations is called dropout. dropping out individual connections is called dropconnect.