# Maximum Likelihood Distribution

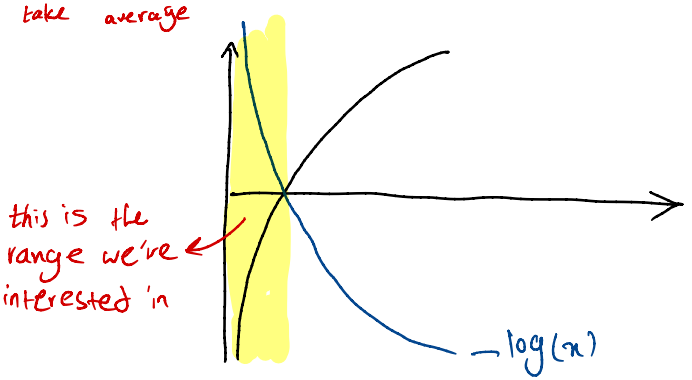
neural nets are typically trained according to Maximum Likelihood Dist because of the loss function we use, aka cross entropy loss

$$J(\theta) = -\mathbb{E}_{x,y \sim \hat{p}_{data}} \log p_{model}(y|x)$$

$\mathbb{E}_{x,y \sim \hat{p}_{data}}$ shows the expected behavior over all (x,y) pairs of data, it just means we take average

$-\log$ turns the probability into a cost function, if the $p_{model}$ is low, $J$ is high

the probability that model predicts $y$, given




a good cost function does not saturate (become flat). if they do, they slow down training

an alternative to cross entropy could be mean absolute error or mean squared error, but they lead to poor result as some outputs saturate and slow the training

the thing about cross entropy is that no model in practice gives $p=0$ or $p=1$, so the minimum is never reached.

a softmax can rarely be used inside a model to choose between various internal options.

cross-entropy and softmax are perfectly fit as CE undoes the log by softmax. other loss functions usually don't work as good, may result in vanishing gradient.

the Softmax can saturate if its inputs are extremely dispropoortionate. this saturation can only be undone by a cost function that can see past the exponentions of softmax.
a more accurate name for softmax is Softargmax.

some functions like ReLU are not totally differentiable. However, in practice this is justified for two reasons:

1. Sparse Non-dif..ility: it rarely happens that we need the grad at a point exactly z=0
2. Two-way diff: implementations use either left or right diff at z=0

the goal of training in practice is not to find points where $\partial L$ is exactly zero, but to minimize.

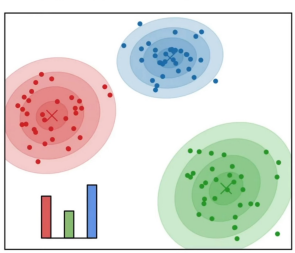
Cross-entropy allows almost any output type

In general, if we define a conditional distribution $p(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta})$, the principle of maximum likelihood suggests we use $-\log p(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta})$ as our cost function.

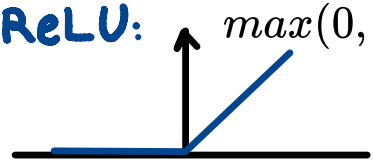
an interesting family of networks
(a bit similar to energy based models?)

**Gaussian Mixture Models**

1. The means
2. The covariance
3. The weight



**ReLU:** $max(0, z)$



the benefit of ReLU that make it very useful:

① 2nd derivative is zero, first derivative is 1 or 0, very simple

② the gradient is large whenever z is large, meaning we don't hit the vanishing gradient typical in sigmoid / tanh.

# Hidden Units (act functions)

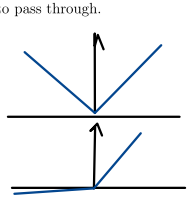
the choice of which act func to use is not really based. default → ReLU
↳ trial & error
↳ experimentation

$$h = g(\boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{b}). \tag{6.36}$$

When initializing the parameters of the affine transformation, it can be a good practice to set all elements of $\boldsymbol{b}$ to a small positive value, such as 0.1. Doing so makes it very likely that the rectified linear units will be initially active for most inputs in the training set and allow the derivatives to pass through.

some ReLU variants:

1. absolute value Rectification good for object recognition
2. Leaky ReLU
3. PReLU: treats $a_i$ as learnable parameter.

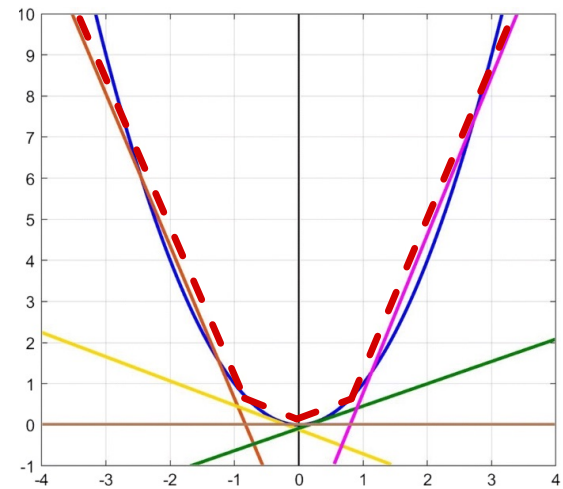


# How can we use AFs that are non-differentiable?

# Maxout Units

divide $z$ into groups and get the max among that group.

$$g(\boldsymbol{z})_i = \max_{j \in \mathbb{G}^{(i)}} z_j,$$




they can approximate any convex function by setting $k \uparrow$.

they introduce more complexity than ReLU.

if the output of various linear layers is fed to maxout, then the output would be k times smaller than concating them.

also better at avoiding catastrophic forgetting.

ReLU and its variants are based on the principle that units which are closer to linearity act better.

# Sigmoid & tanh

Closely related,

for very large and very small $z$ they can saturate → not advised for hidden layers

tanh better than Sigmoid as it is more similar to identity function.

sigmoid could be a better choice in settings other than feedforward networks.

generally, almost all hidden units perform comparably. It's best to test during R&D phase.

# No Activations ?!

Stacking two linear layers results in a linear transformation, yes; but it could also reduce the number of parameters.

$W^T x$ → break down → $V^T U^T x$

$nxp$ params

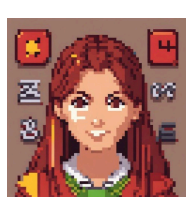
$(n+p)q$ params ($n$, $q$, $p$)

if $W$ is low-rank and $q$ is small, this saves params.

# Architecture Design

⇒ Universal Approximation Theorem: any network with at least one hidden layer can approximate any function by a non-zero error. it does NOT mean that the model "learns" the function, as it may not necessarily generalize, but it can "represent" the model.

In summary, a feedforward network with a single layer is sufficient to represent any function, but the layer may be infeasibly large and may fail to learn and generalize correctly. In many circumstances, using deeper models can reduce the number of units required to represent the desired function and can reduce the amount of generalization error.



# Rule of thumb

deeper networks tend to generalize better. (1)

more depth typically generalize better than more parameters. (2)

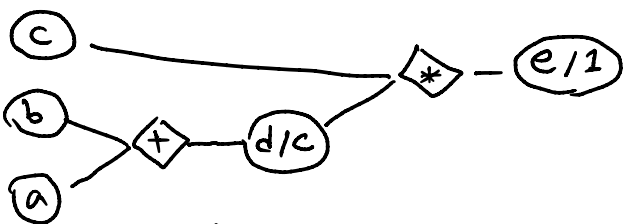
# Backprop

Commonly misused as the entire learning algorithm, but it is the part regarding computing the loss gradient. Another algorithm like the gradient descent is used to train using this gradient of loss.

the gradient we want to compute

$\nabla_\theta J(\theta)$

gradient of loss wrt to the model parameters



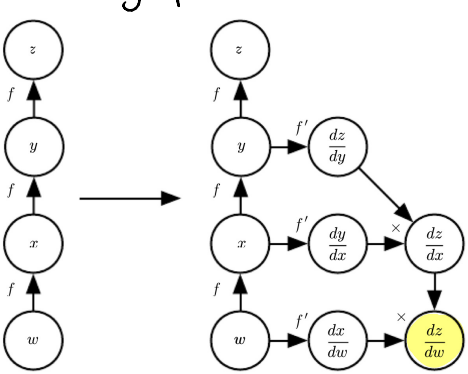

example of a computation graph

## more info on backprop in 0001

Simply computing the gradient by the chain rule directly, imposes challenges. One is many computations are repeated in a graph.

### Symbol-to-Symbol

used by Tensorflow, this creates a second computation graph.



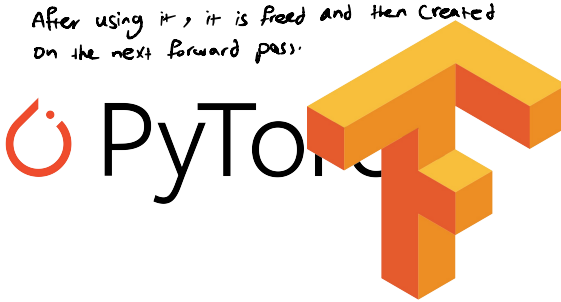

## PyTorch vs. Tensorflow

Tensorflow builds a static graph of the model. Once you define the model and optim and you compile it, TF builds the whole static graph and just reuses it again.

it is fast, but not flexible

Pytorch builds a dynamic graph on the go, during forward pass and uses it once you call the .backward() method.

After using it, it is freed and then created on the next forward pass.




## Deep Learning Renaissance (2000s)

- big datasets available
- more computation
- replacement of MSE by Maximum Likelihood

(MSE and MAE often lead to saturation when used in conjunction with some activations like tanh & sigmoid)

- replacement of Sigmoid hidden units by piece-wise linear hidden units like **ReLU**