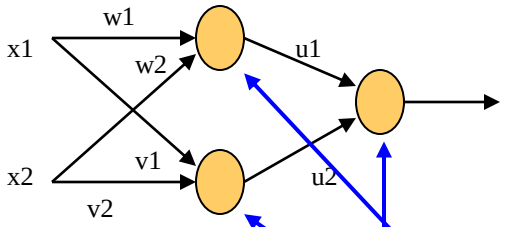
From Shallow to Deep Networks:

tackling the problem of vanishing gradients

- The problem of exploding of vanishing gradients
- Alternative activation functions
- Alternative weight initialization strategies
- Batch Normalization
- Improvements of the SGD algorithm
- Study Chapter 11 of Geron's Book!

Backpropagation (last week)

Consider the network (no biases):



$$net_1 = w_1x_1 + w_2x_2, \quad y_1 = \varphi(net_1)$$

$$net_2 = v_1x_1 + v_2x_2, \quad y_2 = \varphi(net_2)$$

$$net = y_1u_1 + y_2u_2, \quad y = \varphi(net)$$

$$= \varphi(y_1u_1 + y_2u_2) =$$

$$= \varphi(\varphi(net_1)u_1 + \varphi(net_2)u_2) =$$

$$= \varphi(\varphi(w_1x_1 + w_2x_2)u_1 + \varphi(v_1x_1 + v_2x_2)u_2)$$

Recursive Delta Rule => v/e gradients

$$\frac{\partial E}{\partial u_1} = (y - d)\phi'(net)y_1$$

$$\frac{\partial E}{\partial w_1} = (y - d)\underline{\phi'(net)}u_1\underline{\phi'(net_1)}x_1$$

… (see the generalized delta-rule)

The deeper you go the more multiplications of gradients ($\phi'(net)$) you have!

Products of small numbers is very small!
Products of big numbers is a very big!

The problem of exploding or vanishing gradients!

General case: Generalized Delta Rule

$\Delta w_{ji} = \eta \delta_j y_i$, where:

$$\delta_j = \begin{cases} \varphi'(v_j)(d_j - y_j) & \text{IF j output node} \\ \varphi'(v_j) \sum_{k \text{ of next layer}} \delta_k w_{kj} & \text{IF j hidden node} \end{cases}$$

w_{ji} : *weight from node j to node i (moving from output to input!)*

η : *learning rate (constant)*

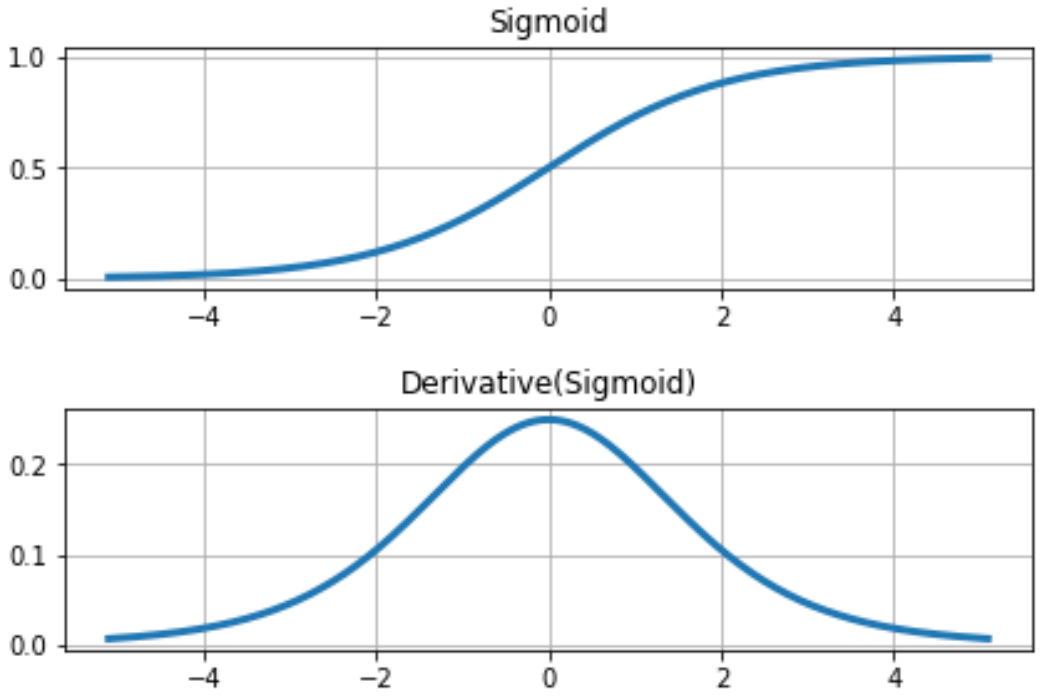
y_j : *output of node j*

v_j : *activation of node j*

Fix 1: Alternative activation functions

- Logistic
- Sigmoid
- ReLU (Rectified Linear Unit) (most popular!)
- Leaky ReLU
- ELU (Exponential Linear Unit)
- SELU (Scaled ELU) *(“magic”;)*
- …
- Soft-max (applied to a layer)
- Check: https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html

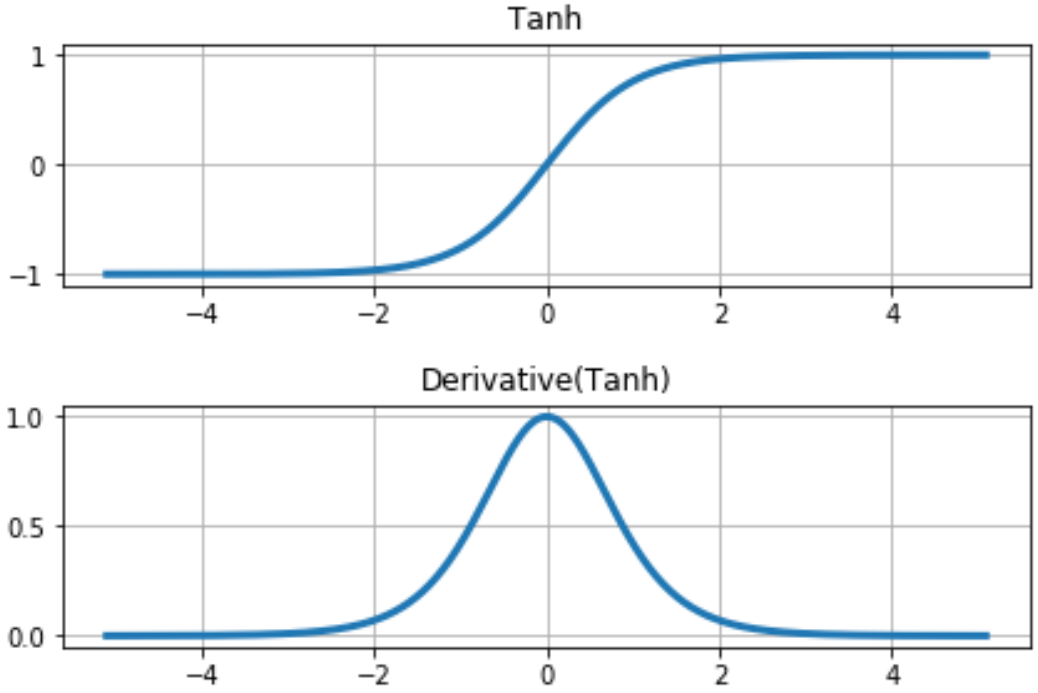
Logistic (sigmoid) function



- Most popular till ~2010's
- Saturates around -7 or +7 (?)
- Derivative at most 0.25
- Derivative vanishes when the function saturates

=> vanishing gradients

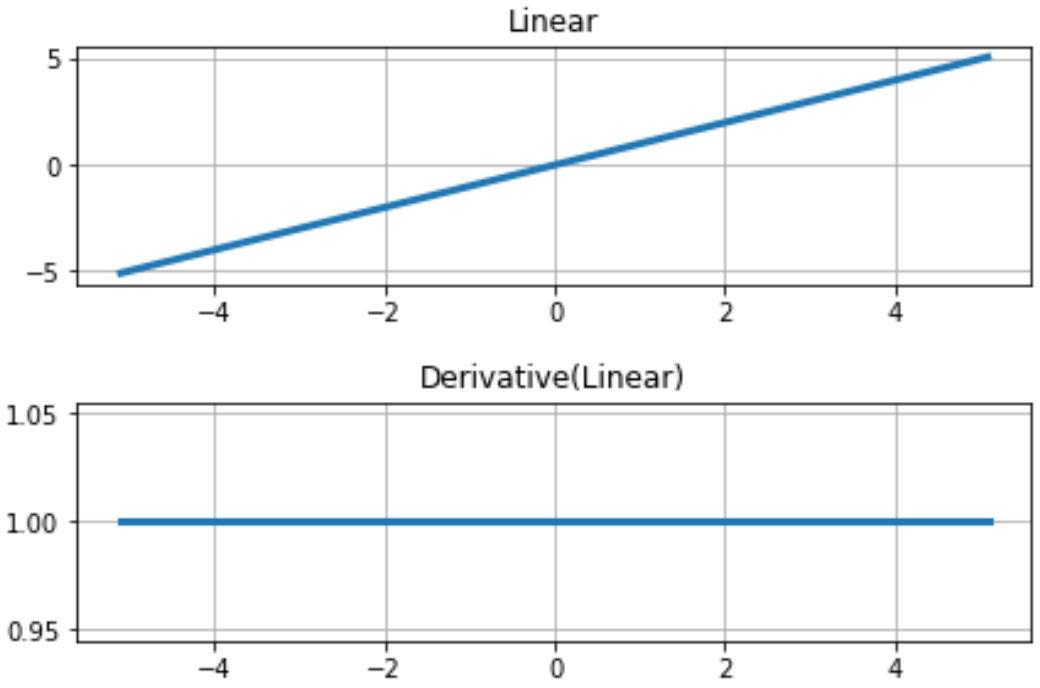
TanH function



- Most popular till ~2010's
- Quickly saturates
- Derivative at most 1.0
- Derivative vanishes when the function saturates

=> vanishing gradients

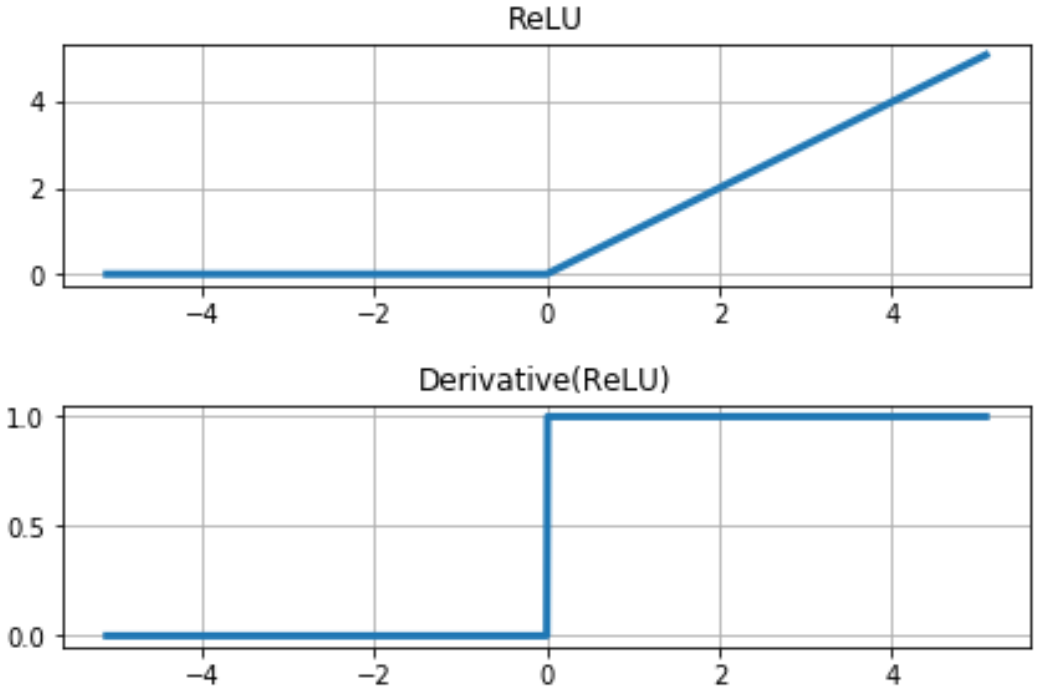
Linear (identity) function



- E.g. output node in regression tasks
- Derivative is constant (1.0)
- Using this function in all layers doesn’t make sense

=> a combination of linear functions is a linear function!

Rectified Linear Unit (ReLU)

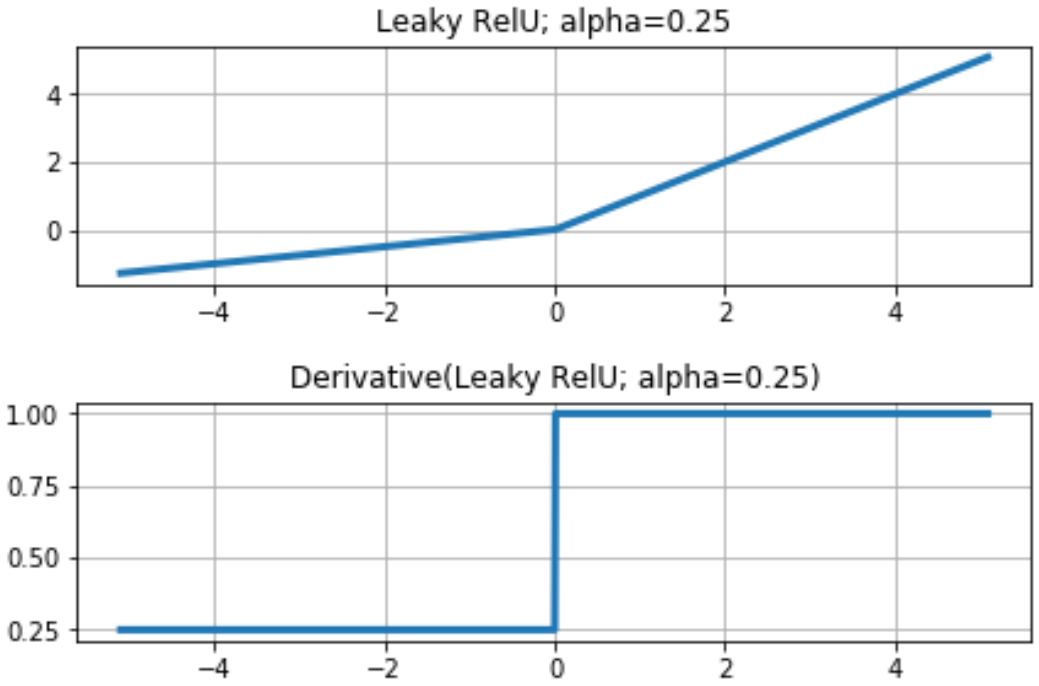


- Most popular since 2012 (AlexNet; A. Kryzevski)
- Derivative 0.0 or 1.0
- Super fast to compute both f(x), f'(x)), w. update

+ no multiplications, just "sign tests"

- "dying neurons, when 0"

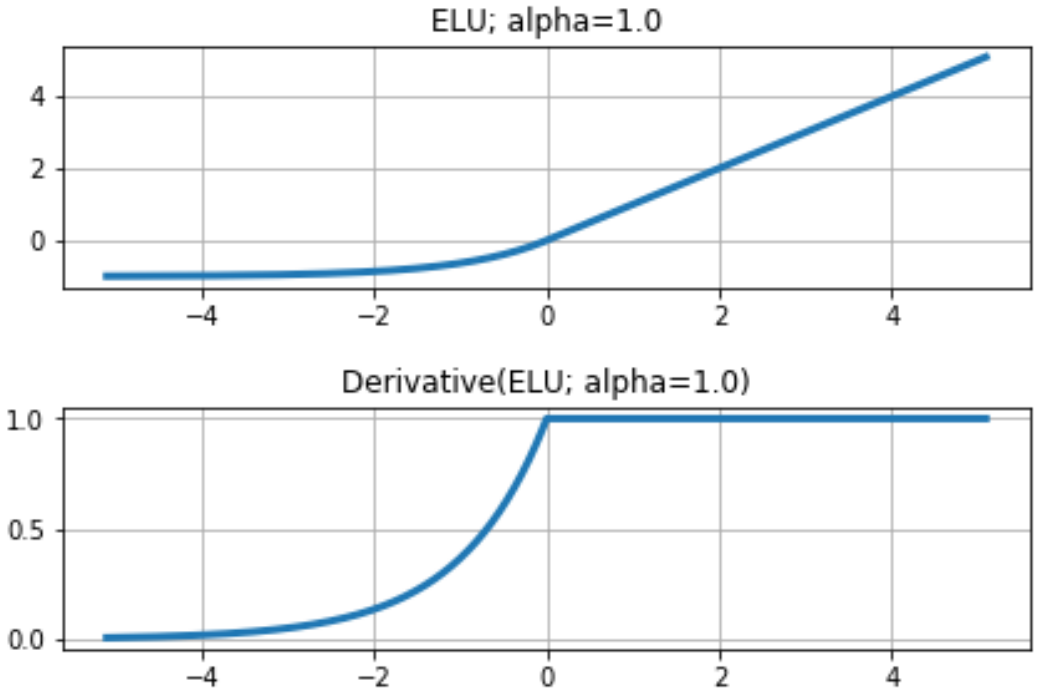
Leaky Rectified Linear Unit (LReLU)



- Very popular
- Derivative = *alpha* or *1.0*
- Avoids the vanishing gradients problem

=> At most 1 multiplication, and just one “sign tests”

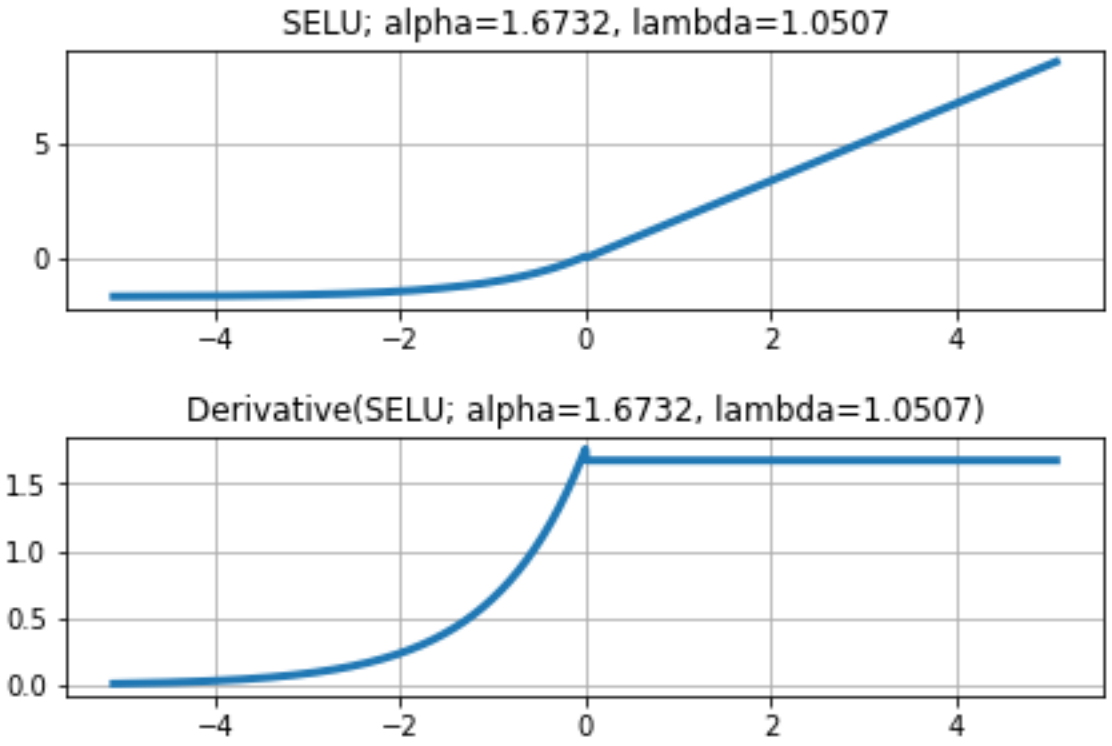
Exponential Linear Unit (ELU)



- A “smooth” variant of LReLU
- Derivative vanishes very slowly
- One non-tunable parameter (*alpha*)

=> Better than LeakyReLU

Scaled Exponential Linear Unit (SELU)



- Two non-tunable parameter (*alpha, lambda*)
- For specific values of these parameters: *“self-stabilization”*

=>preserves variance of activations when passing layers

Details:

https://towardsdatascience.com/selu-make-fnns-great-again-snn-8d61526802a9

Conclusions

- Many variants of activation functions
- The choice of the “right” activation function depends on:
  - the type of data (images? sequences? vectors? ..)
  - the network architecture (MLP, CNN, RNN, AE, …)
  - the task (Classification? Regression? Image segmentation? Feature extraction? …)
  - …

=> Get inspiration by studying relevant papers!

Weights Initialization Strategies

- Xavier Glorot and Joshua Bengio (2010): Traditional weight initialization strategies (for logistic activation functions):
  - standard Gaussian (mean=0, σ =1)
  - uniform (on [-1, 1])

have a nasty property: variance of outputs (of single nodes/layers) increases, so the deeper we go the bigger the variance!

(**Q:** what about initializing all weights to the same value? Eg., to 0?)

Initialization strategies

- For each layer calculate:

  fan_{in}=number of connections “to” the given layer,
  fan_{out}=number of connection “from” the given layer
  $fan_{avg}=(fan_{in}+fan_{out})/2$

- Then generate weights (for each layer) from a normal distribution with 0 mean and:

  (Glorot): for logistic, tanh, softmax: $\sigma^2 = 1/fan_{avg}$
  (He): for ReLU and its variants: $\sigma^2 = 2/fan_{in}$
  (LeCun): for SELU: $\sigma^2 = 1/fan_{in}$

*A layer: **n** neurons, each with **k** inputs: **fan_{in}=k**; **n** outputs: **fan_{out}=n** (bias?)*

Progress

- By introducing “alternative” activation functions and weight initialization strategies it was possible to train networks with 10-20 layers.

- Another big jump forward was the invention of the **Batch Normalization** algorithm (Joffe & Szegedy, 2015): up to 100 layers!

  Check the original paper: https://arxiv.org/pdf/1502.03167.pdf

- A yet another jump was achieved with allowing “shortcuts” between layers (**ResNet**, He, Zhang, Ren, 2015): up to 1000 layers!

Batch Normalization: key idea

- When training a network with batches of data (the most common procedure!) the network "gets confused" by the fact that statistical properties of batches are varying from batch to batch (when we look at it at an arbitrary, 'fixed' layer)
  - A batch: a matrix #records x #variables (batch_size x #dimensions)
  - Statistical properties: **the mean** (a vector of means) and **the std** (a vector of standard deviations)
- Idea 1: Normalize each batch => subtract the mean and divide by the std deviation.
- Idea 2: Assume that it is beneficial to scale and to shift each batch by certain γ, β, to minimize network loss (error) on the whole training set.
- Idea 3: Finding optimal γ, β, can be achieved with SGD!

Batch Normalization: formulas …

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

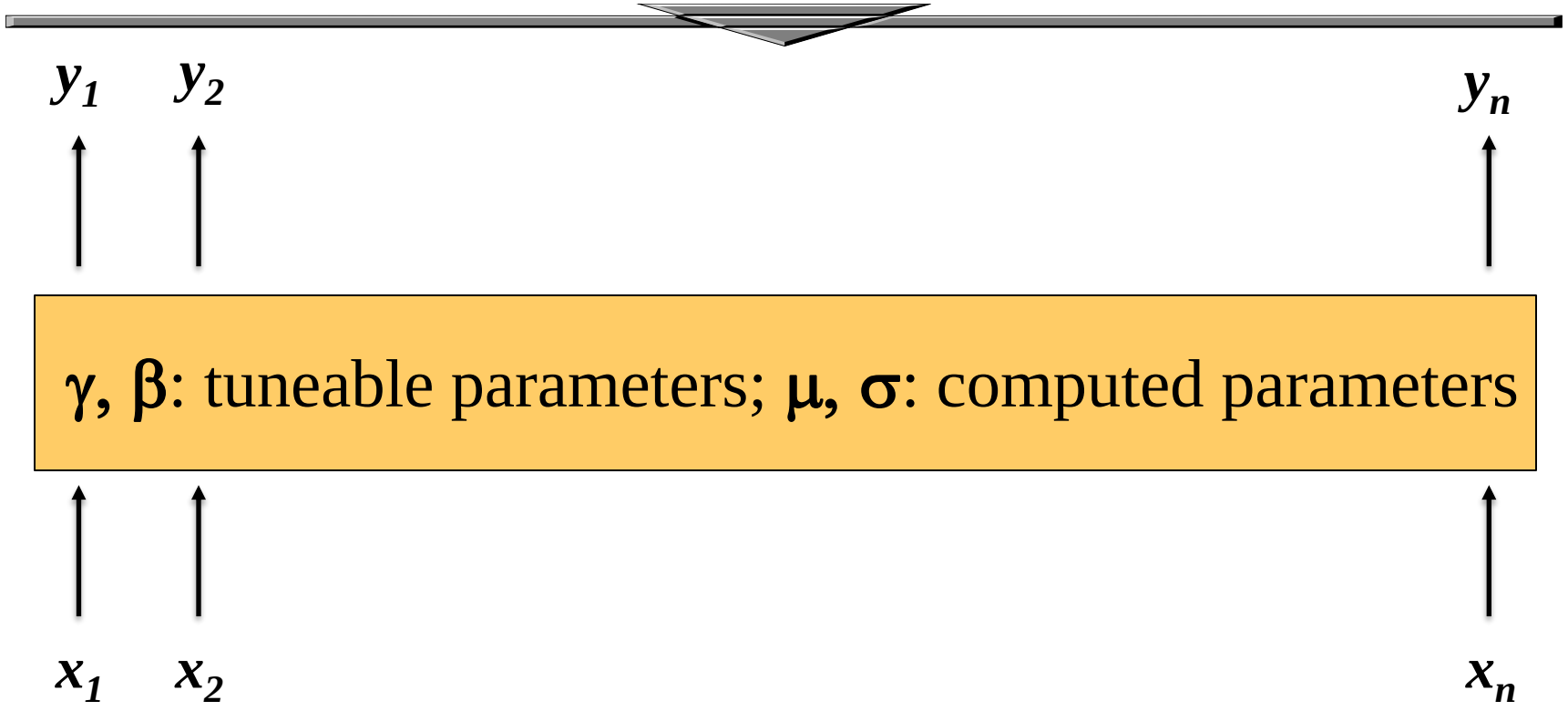
$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

- Batch of size m records; x_i – vectors, ε: to prevent *sth/0* problem.
- γ and β : parameters to be optimized (each of the dimensionality of x_i).

Batch Normalization Layer



- How to propagate gradients of the loss function?
- Apply the chain rule to the definition of the BN transformation!

Computing gradients of BN transformation

$$\frac{\partial \ell}{\partial \hat{x}_i} = \frac{\partial \ell}{\partial y_i} \cdot \gamma$$

$$\frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial \hat{x}_i} \cdot (x_i - \mu_{\mathcal{B}}) \cdot \frac{-1}{2}(\sigma_{\mathcal{B}}^2 + \epsilon)^{-3/2}$$

$$\frac{\partial \ell}{\partial \mu_{\mathcal{B}}} = \left(\sum_{i=1}^{m} \frac{\partial \ell}{\partial \hat{x}_i} \cdot \frac{-1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \right) + \frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{\sum_{i=1}^{m} -2(x_i - \mu_{\mathcal{B}})}{m}$$

$$\frac{\partial \ell}{\partial x_i} = \frac{\partial \ell}{\partial \hat{x}_i} \cdot \frac{1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} + \frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{2(x_i - \mu_{\mathcal{B}})}{m} + \frac{\partial \ell}{\partial \mu_{\mathcal{B}}} \cdot \frac{1}{m}$$

$$\frac{\partial \ell}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial y_i} \cdot \hat{x}_i$$

$$\frac{\partial \ell}{\partial \beta} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial y_i}$$

- Batch of size m records; x_i – vectors, ε: to prevent problem
- γ and β: parameters to be optimized (each of the dimensionality of x_i)

Batch Normalization: test mode

- While training, every batch is normalized separately but the parameters β and γ are tuned to be optimal for all batches. But in the test mode we don't know how to normalize incoming data! We need μ and σ over the whole training set!

- Solution 1 (very expensive! **Q: WHY? (homework!)**):

  Treat the whole training set as one big batch:
  for each BN layer compute μ and σ and use them for testing.

- Solution 2 (Keras' default):

  To estimate μ and σ process batch after batch, applying the moving average "trick";
  then use the final estimates for testing.

Exponential Moving Average: set $\mu_{1=}\mu_{B1}$ and then: $\mu_{n+1}= 0.01*\mu_{Bn+1} + 0.99*\mu_n$

for all batches; the same procedure for estimating global σ's.

You have it for free!

BN: bigger learning rates => less iterations

Batch Normalization allows higher learning rates (normalization prevents saturation or weights), reducing the number of epochs; consequently, it is much faster than other training algorithms!

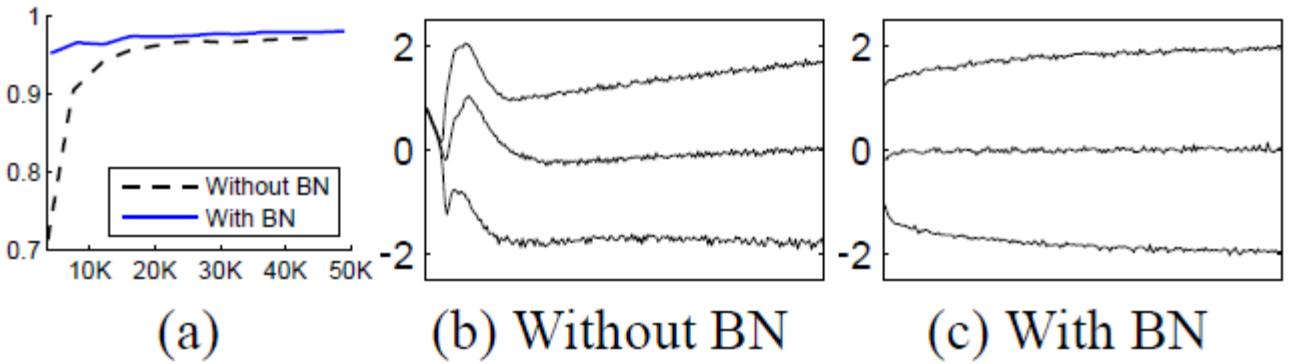


Figure 1: (a) *The test accuracy of the MNIST network trained with and without Batch Normalization, vs. the number of training steps. Batch Normalization helps the network train faster and achieve higher accuracy.* (b, c) *The evolution of input distributions to a typical sigmoid, over the course of training, shown as* $\{15, 50, 85\}$*th percentiles. Batch Normalization makes the distribution more stable and reduces the internal covariate shift.*

BN: Superior Accuracy and Speed

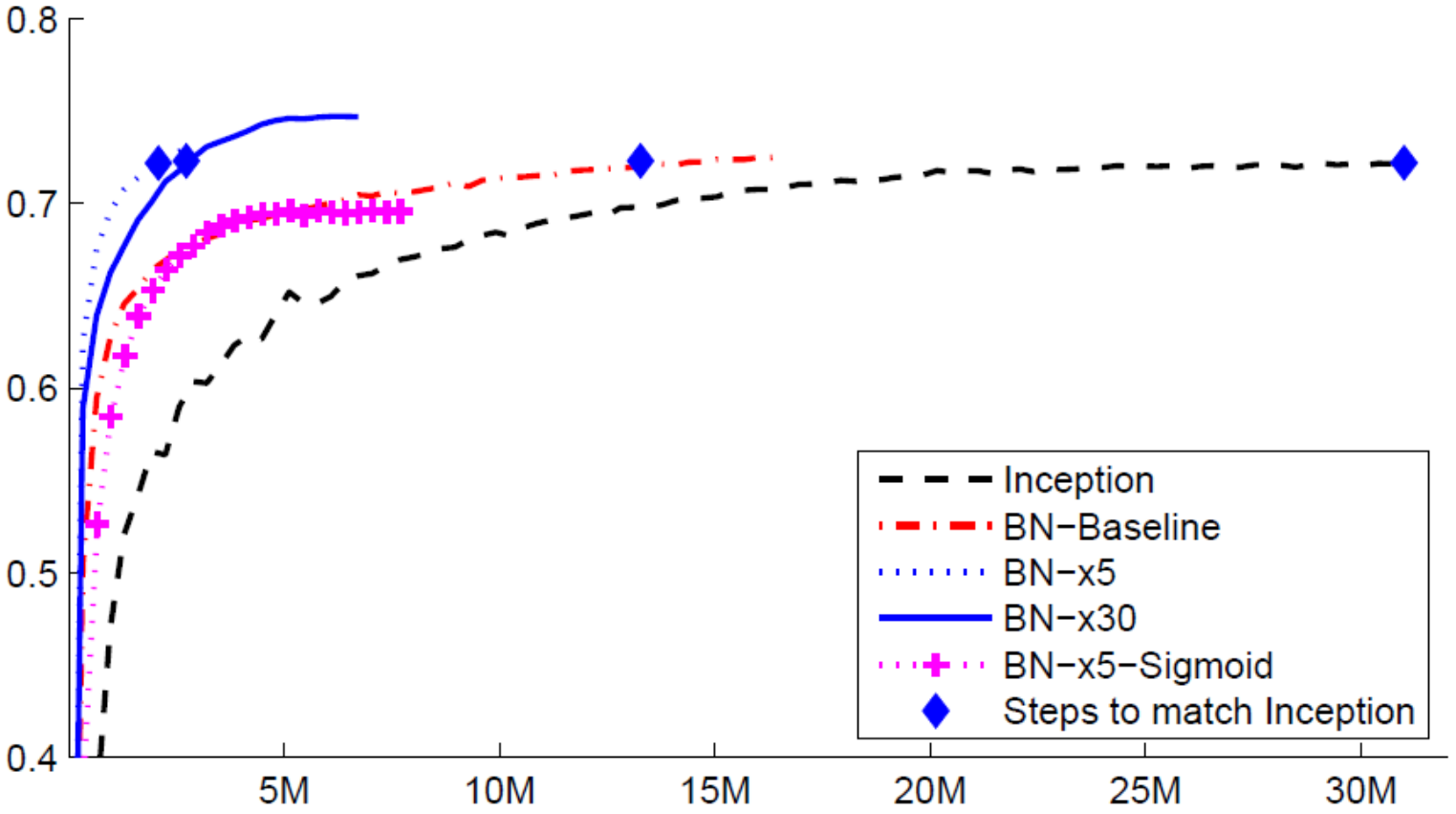


Figure 2: *Single crop validation accuracy of Inception and its batch-normalized variants, vs. the number of training steps.*

Conclusions BN

- Superior accuracy.
- Reduces the risk of vanishing/exploding gradients.
- Much faster (in terms of time and #iterations) than the traditional BP.
- Allows for using “big learning rates” => less epochs needed for convergence.
- Allows for training much deeper networks (100 layers?)
- BN also increases regularization: lower risk of overfitting.

Regularization Methods

- More weights/nodes => higher chance of **data overfitting**
- **Regularization**: an additional mechanism to prevent overfitting
- **L1** or **L2 regularization**: penalty on too big values of weights
- **Alternative loss functions**, MSE, cross-entropy, LogLoss, HuberLoss, …
- **Batch Normalization**
- **Node Dropout**: at every training step (processing a batch), each node (not from the output layer) has a chance 0<p<1 to be disabled; when testing all nodes are active (a form of bagging a collection of networks)
- **Monte Carlo Dropout**: normal Dropout + “when testing, aggregate predictions over random sub-networks”

When training a network we have a specific “accuracy” or “error” measure in mind. However, our “true objective” is often reformulated in terms of a “loss function”. Thus training “optimizes the loss function”, while our true “accuracy” might be quite different. ***Be aware of it!***

Variants of SGD optimization

- Gradient Descent (learning rate)
- Gradient Descent with Momentum
- Nesterov Accelerated Gradient
- Adaptive Gradient Descent (AdaGrad)
- RMSProp
- Adam (adaptive moment estimation)
- Nadam
- Etc.