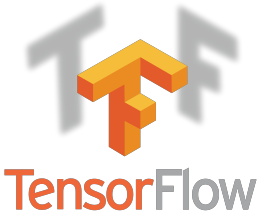
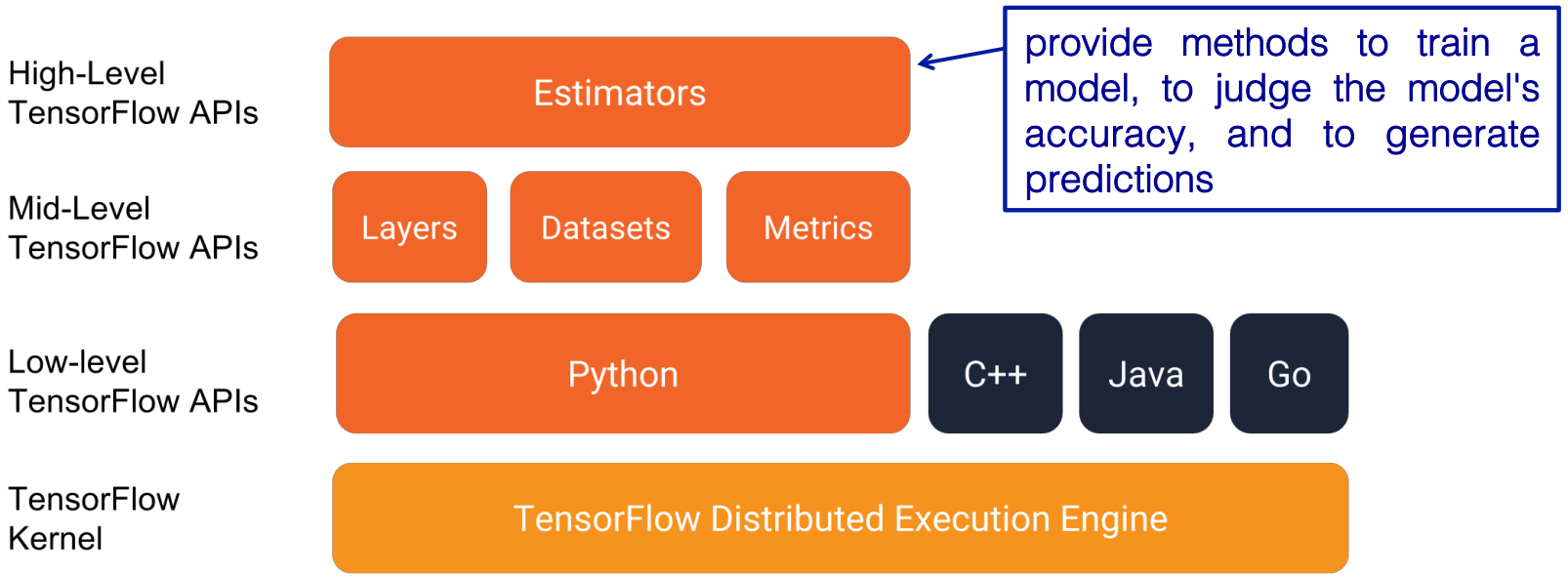
TensorFlow



- TensorFlow™ is an open source software library for numerical computation using data flow graphs (computational graph)
- Nodes in the graph represent mathematical operations
- Graph edges represent the multidimensional data arrays (tensors) communicated between nodes
- With a single API it allows to deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device



Warning!

We are going to use Python

If you do not know Python, here is a great tutorial tailored for our aims

http://cs231n.github.io/python-numpy-tutorial/

Goals

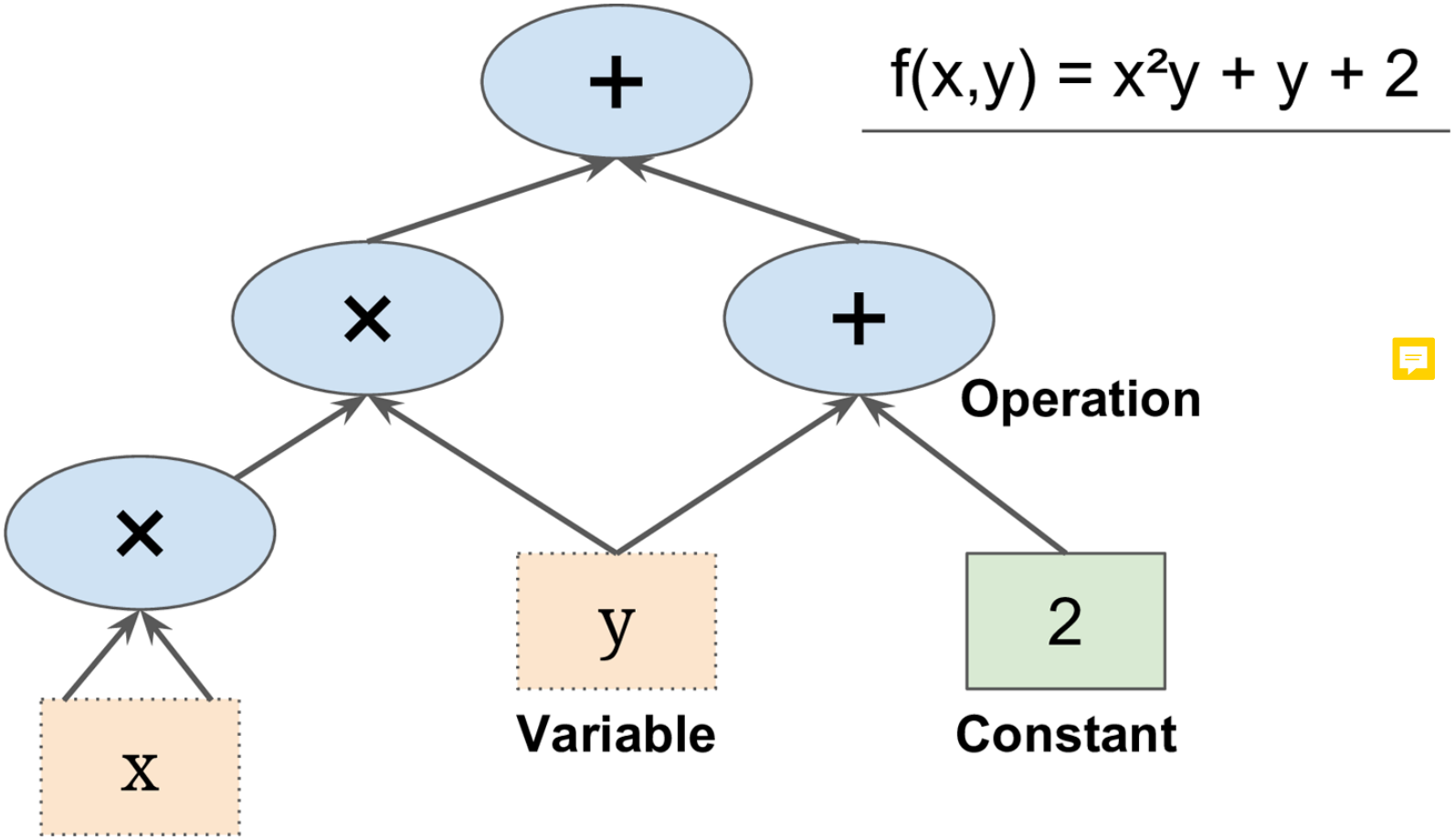
We will try to:

- understand TF's basic concepts underpinning the computational graph approach
- explore some of the built-in functions
- learn how to build a pipeline for building deep learning models

Covered Topics

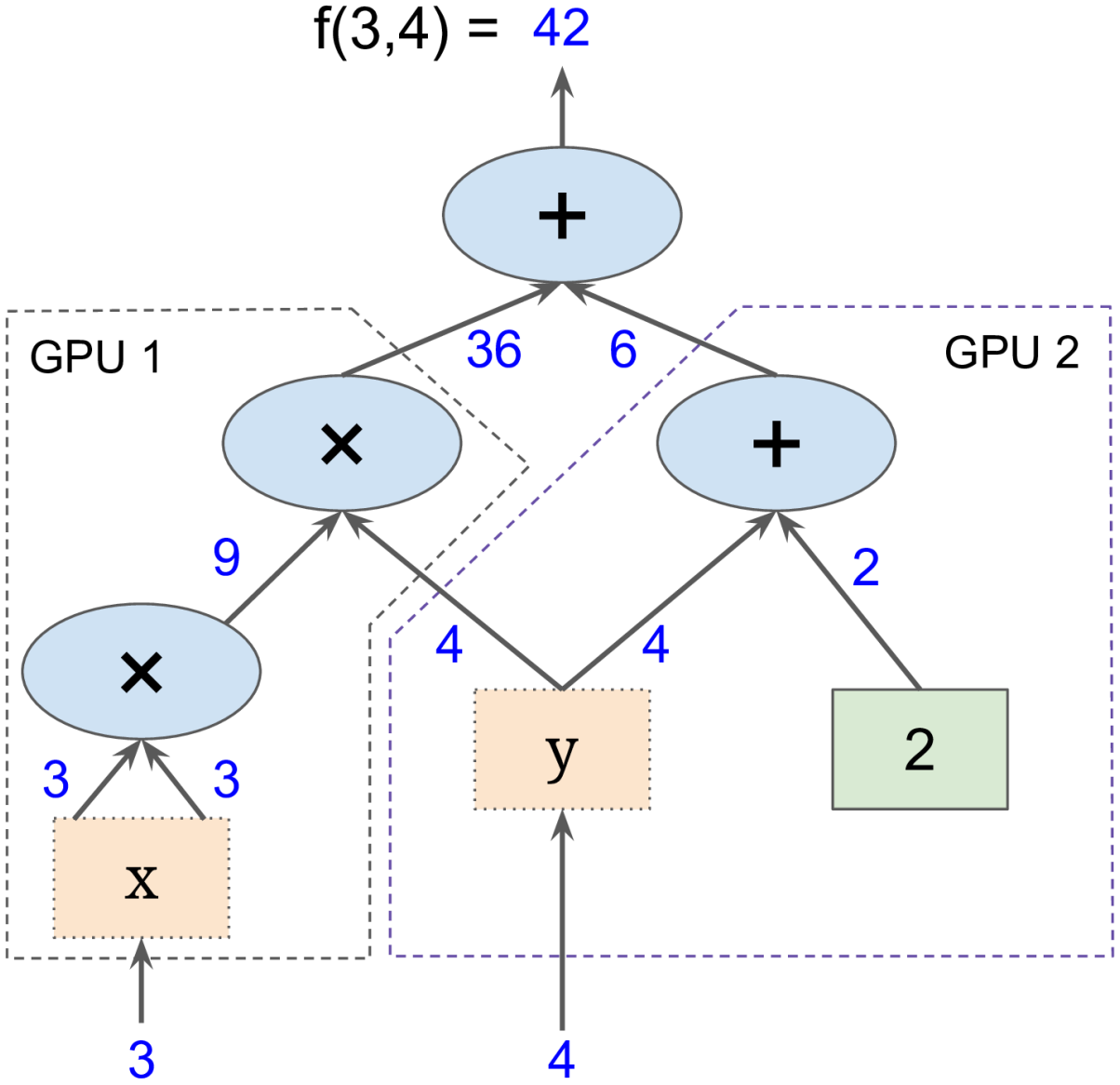
- Computational graph in TF and Reverse-Mode Autodiff
- Mini-batches
- Weights initialization
- Alternative units: Leaky ReLU, ELU, SELU
- Batch Normalization
- Nesterov Accelerated Gradient
- Dynamic Adaptation of Learning Rates
- Regularization:
    - Early Stopping
    - Extended Loss
    - Dropout

Computation Graph Construction



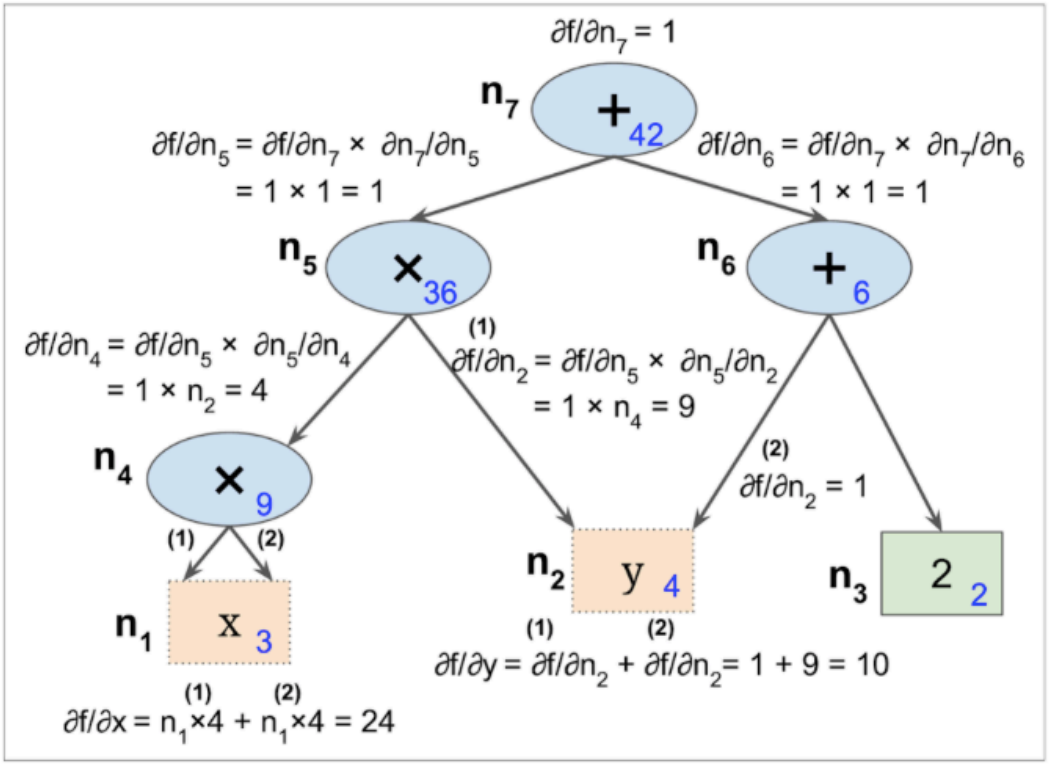
$$f(x,y) = x^2y + y + 2$$

Computation Graph Execution



Reverse-Mode Autodiff

1. Forward direction (from input to output) to compute the value of each node
2. Reverse direction (from output to input) to compute all the partial derivatives



Requires one forward and one reverse pass per output

Very efficient with many inputs and few outputs

Mini-batches

Full gradient descent computes the gradient of the Loss on the WHOLE training set and THEN updates the weights

*Pro: Loss depends on ALL examples, it computes the **right** gradient*

Con: if training set contains 1 million examples, only 1 weights update per epoch (i.e. 1 update after having computed the output for ALL examples)

Stochastic gradient descent computes the gradient of the Loss on a SINGLE example of the training set and then updates the weights

Pro: if training set contains 1 million examples, 1 weights update per example (i.e. 1 million updates per epochs)

Con: gradient descent is less accurate since it computes the gradient of Loss ONLY with respect to a single example

COMPROMISE

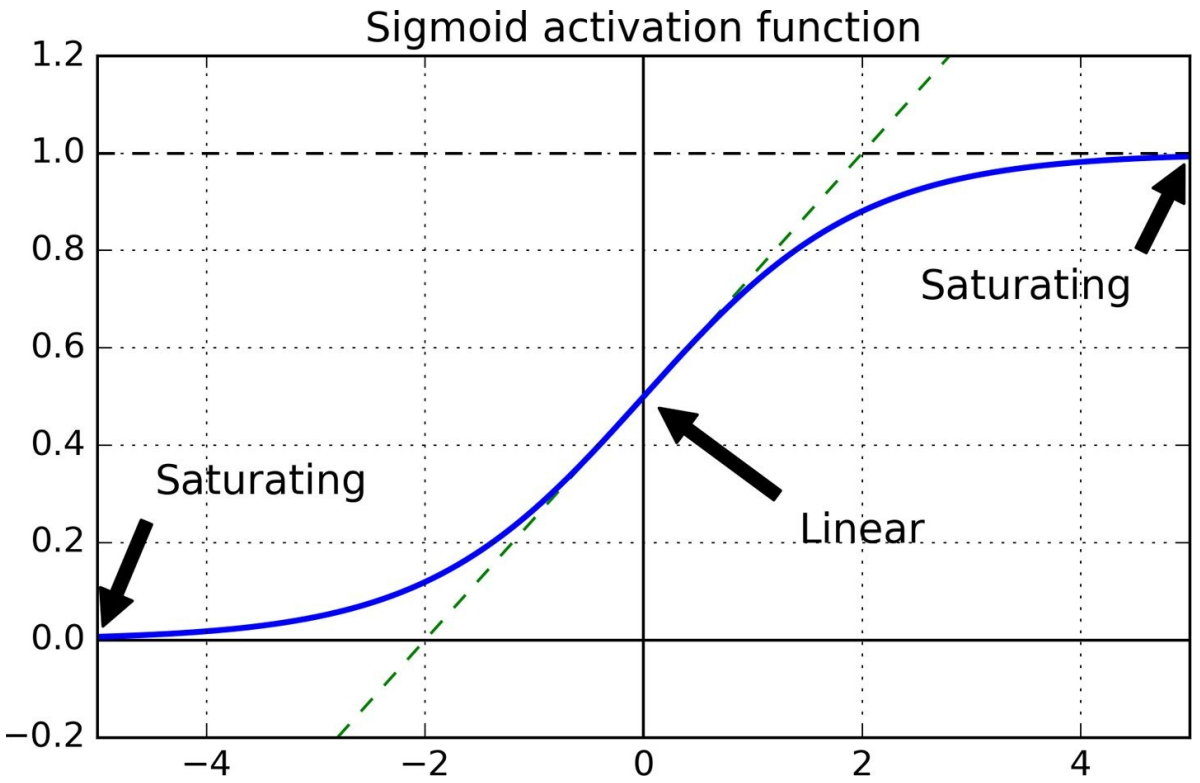
Mini-batch gradient descent computes the gradient of the Loss on a subset of examples (mini-batch) of the training set and then updates the weights

trade-off: more accurate computation of the gradient vs reduced number of updates

moreover: - allows for distribution of computation on several computational devices

- can leverage matrix/matrix operations, which are more efficient (e.g., GPUs)

Saturation of sigmoid function



Need for careful initialization, not only for sigmoidal functions

Weights Initialization

- the variance of the outputs of each layer should be equal to the variance of its inputs
- the same should hold for the gradient
- trade-off: for each unit normalize the initial weights with respect to the number of incoming and outcoming connections

Examples of distributions (He inizialization) for inizialization according to the type of activation function (Normal distribution: mean = 0)

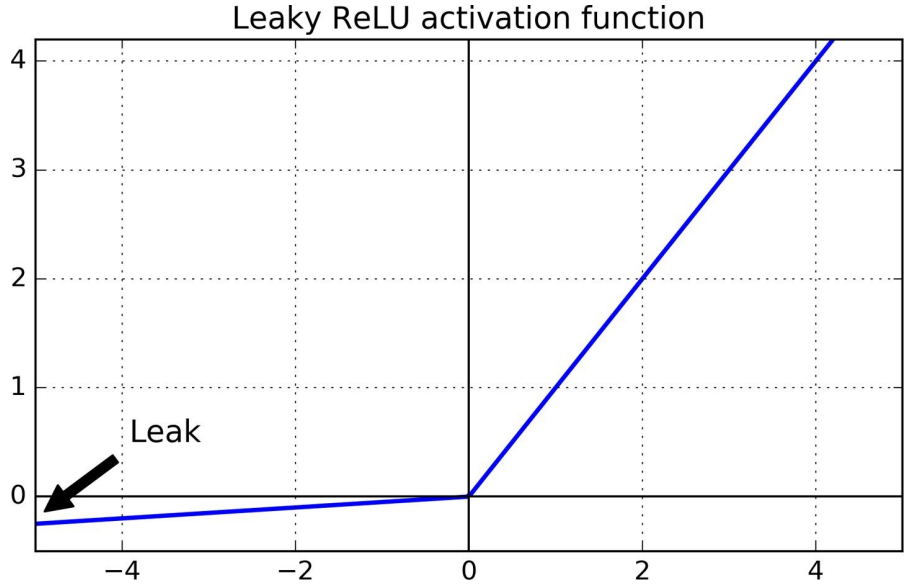
Activation function	Uniform distribution [–r, r]	Normal distribution
Logistic	$r = \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
Hyperbolic tangent	$r = 4\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = 4\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
ReLU (and its variants)	$r = \sqrt{2}\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{2}\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$

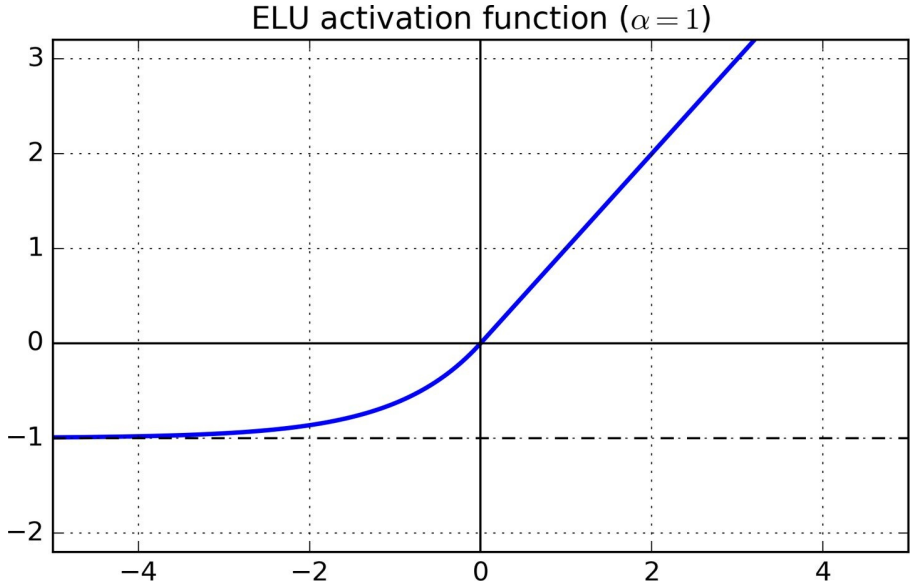
ReLU, Leaky ReLU, ELU

Problem with ReLU: gradient of negative pre-activation is 0
Consequence: ReLU units get stuck to 0 output (i.e., they die!)

Proposed alternatives

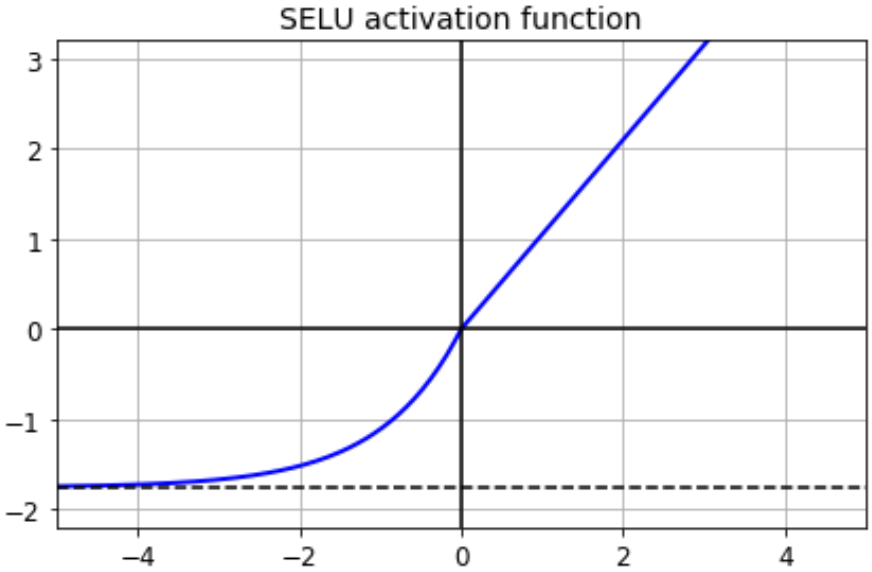
$$\mathrm{ELU}_\alpha(z) = \begin{cases} \alpha(\exp(z) - 1) & \text{if } z < 0 \\ z & if z \geq 0 \end{cases}$$





A New Function: SELU

https://arxiv.org/pdf/1706.02515.pdf



+ normalization: 0 mean, 1 stddev

Batch Normalization

- LeCun et al. (1998) showed that normalizing the inputs speeds up training
- Ioffe and Szegedy (2015) proposed Batch Normalization to normalize hidden (pre-)activations
  - each unit's pre-activation is normalized (mean subtraction, standard deviation division)
  - during training, mean and standard deviation is computed for each mini-batch
  - backpropagation takes into account the normalization
  - at test time, the global mean and global standard deviation is used

Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

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

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

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

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

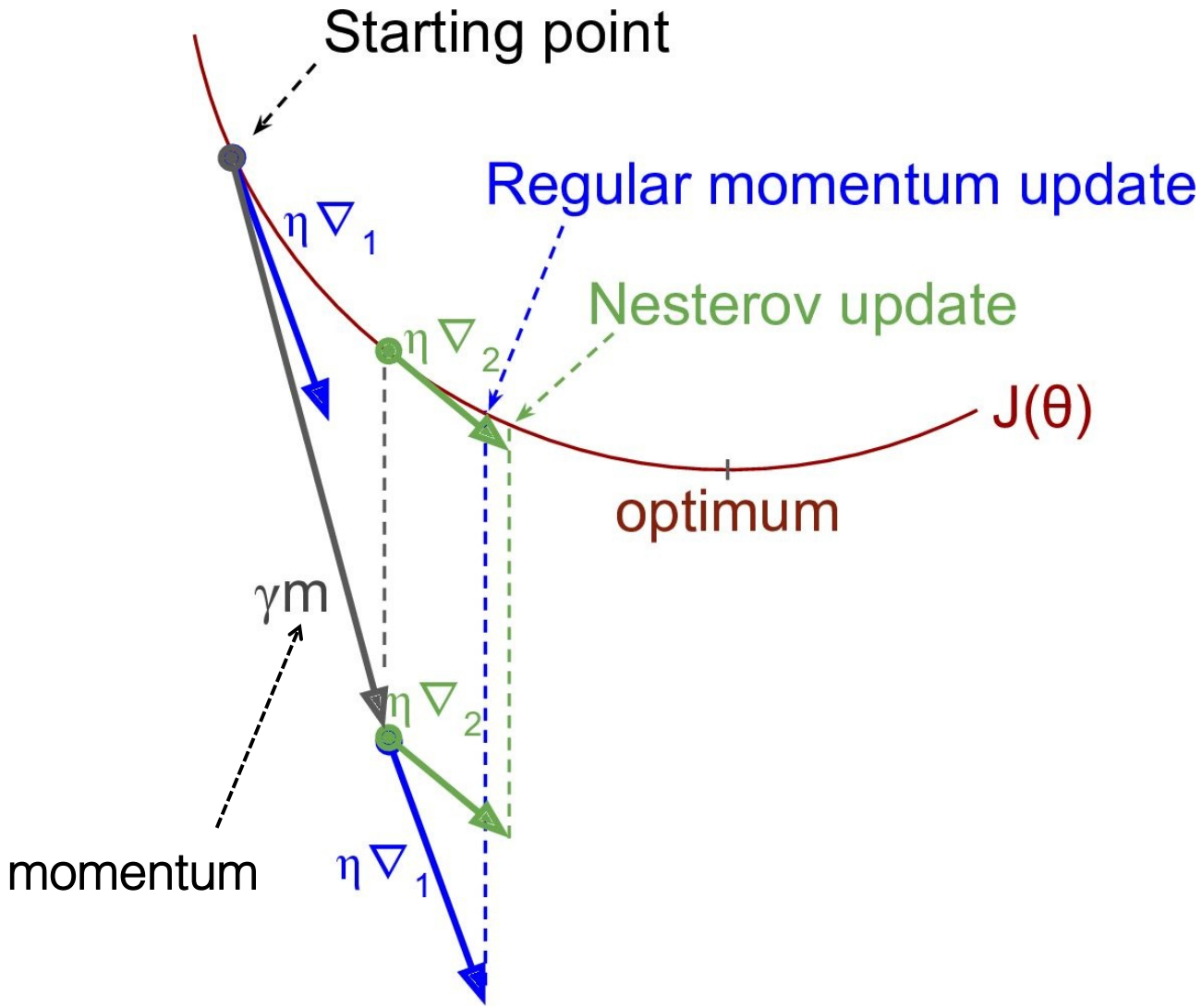
Learned linear transformation to adapt to non-linear activation function (γ and β are trained)

Batch Normalization

- Why normalize the pre-activation ?
  - can help to keep the pre-activation in a non-saturating regime

- At test time use the global mean and global standard deviation
  - removes the stochasticity of the mean and standard deviation
  - requires a final phase where, from the first to the last hidden layer:
    - propagate all training data to that layer
    - compute and store the global mean and global standard deviation for each unit
  - a running average can be used for early stopping

Nesterov Accelerated Gradient



Dynamic Adaptation of Learning Rates

- Updates with adaptive learning rates ("one learning rate per parameter")

  - Adagrad: learning rates are scaled by the square root of the cumulative sum of squared gradients

$$\gamma^{(t)} = \gamma^{(t-1)} + \left(\nabla_\theta l(\mathbf{f}(\mathbf{x}^{(t)}), y^{(t)})\right)^2 \quad \overline{\nabla}_\theta^{(t)} = \frac{\nabla_\theta l(\mathbf{f}(\mathbf{x}^{(t)}), y^{(t)})}{\sqrt{\gamma^{(t)} + \epsilon}}$$

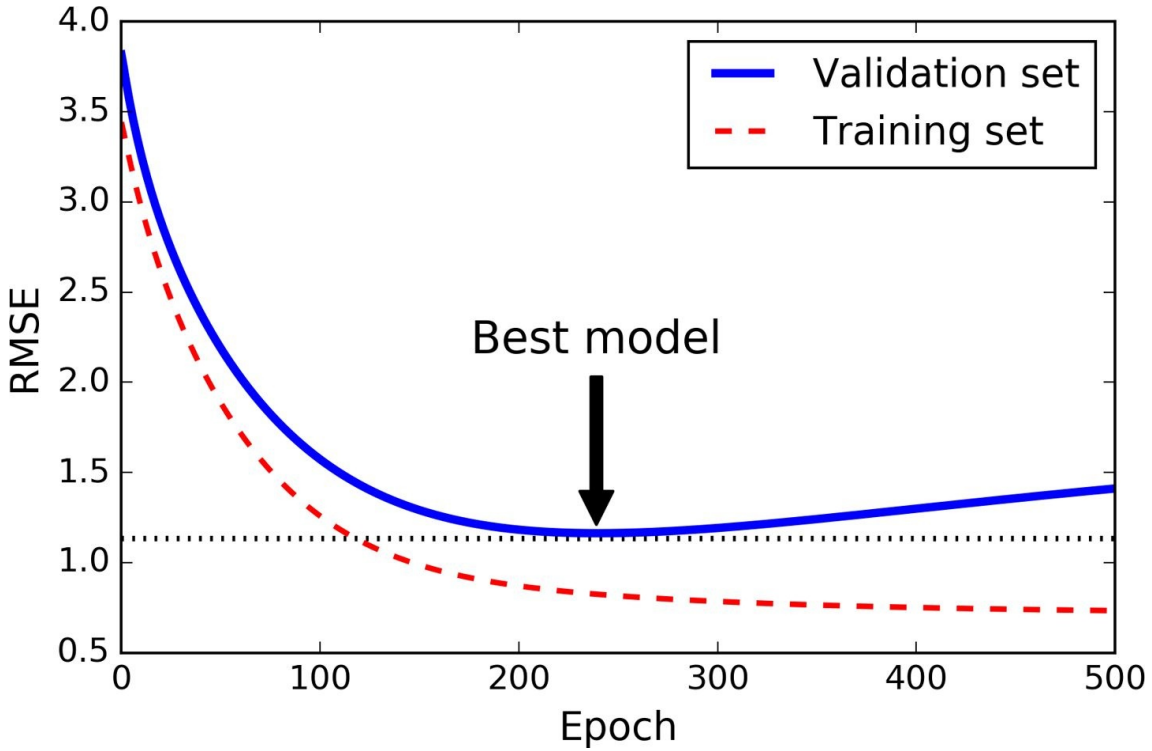
  - RMSProp: instead of cumulative sum, use exponential moving average

$$\gamma^{(t)} = \beta\gamma^{(t-1)} + (1-\beta)\left(\nabla_\theta l(\mathbf{f}(\mathbf{x}^{(t)}), y^{(t)})\right)^2$$

  - Adam: essentially combines RMSProp with momentum

$$\overline{\nabla}_\theta^{(t)} = \frac{\nabla_\theta l(\mathbf{f}(\mathbf{x}^{(t)}), y^{(t)})}{\sqrt{\gamma^{(t)} + \epsilon}}$$

Regularization with Early Stopping



- evaluate the model on a validation set at regular intervals
- store the current best performing model (on validation)
- stop if the best stored model is not changed for many epochs
- use the saved model

Regularization with Extended Loss

Actual Learning Problem:

$$\vec{w}^{\star} = \arg\min_{\vec{w}} \underbrace{J[\vec{w}]}_{\text{Loss}^{\star}} + \lambda \underbrace{\Omega[\vec{w}]}_{\text{Regularizer}}$$

Thus, we need to compute $\nabla\Omega[\vec{x}]$ as well

Examples of regularizers

- L2 regularization

$$\Omega[\vec{w}] = \sum_i w_i^2 = \|\vec{w}\|_F^2$$

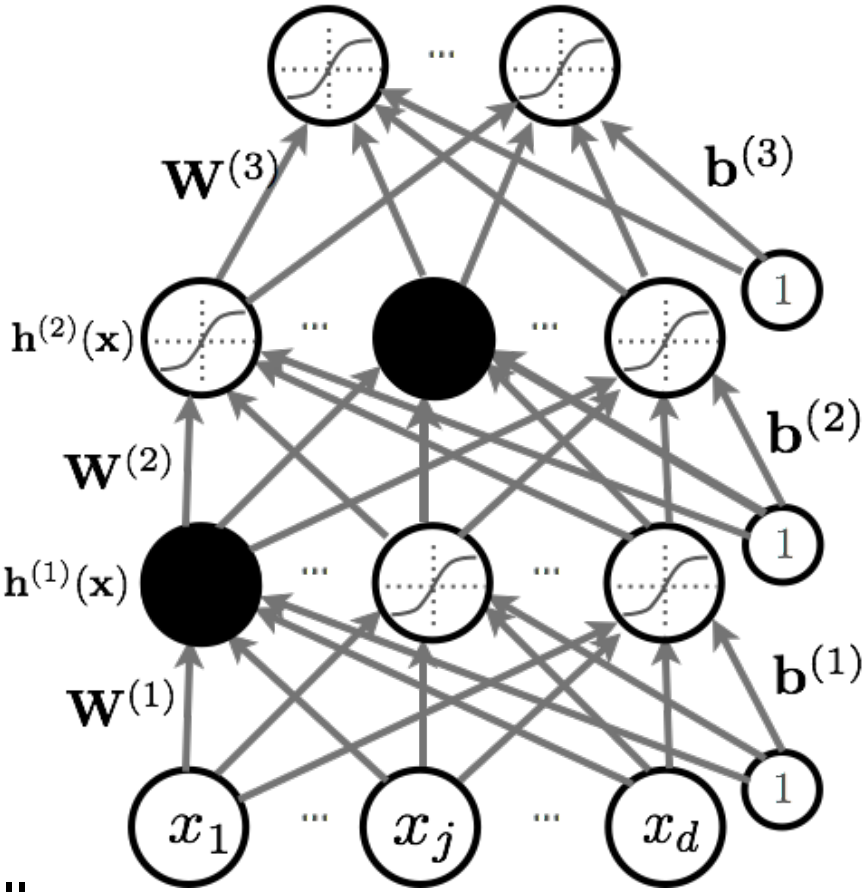
- L1 regularization

$$\Omega[\vec{w}] = \sum_i |w_i|$$

Regularization with Dropout

• Key idea: Cripple neural network by removing hidden units stochastically

- each hidden unit is set to 0 with probability 0.5
- hidden units cannot co-adapt to other units
- hidden units must be more generally useful



• Could use a different dropout probability, but 0.5 usually works well

N. Srivastava et al.; JMLR 15(Jun):1929-1958, 2014

Regularization with Dropout

- Use random binary masks $\mathbf{m}^{(k)}$
  - layer pre-activation for k>0

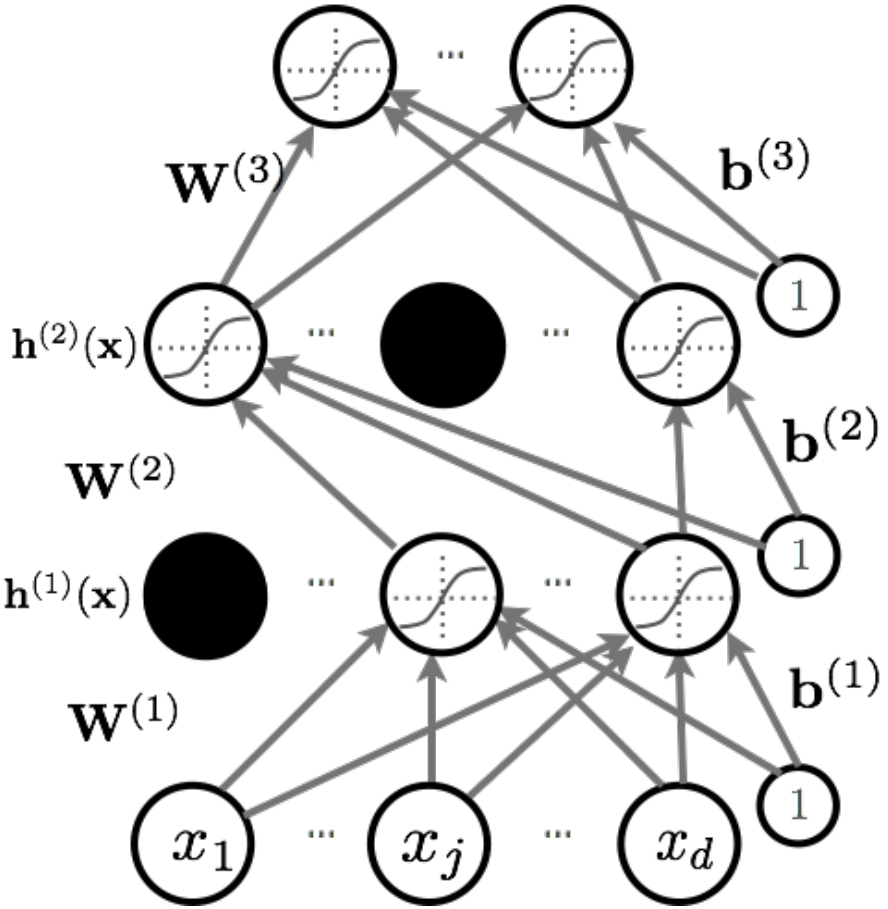
$$\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{b}^{(k)} + \mathbf{W}^{(k)}\mathbf{h}^{(k-1)}(\mathbf{x})$$

  - hidden layer activation (k=1 to L):

$$\mathbf{h}^{(k)}(\mathbf{x}) = \mathbf{g}(\mathbf{a}^{(k)}(\mathbf{x})) \odot \mathbf{m}^{(k)}$$

  - Output activation (k=L+1)

$$\mathbf{h}^{(L+1)}(\mathbf{x}) = \mathbf{o}(\mathbf{a}^{(L+1)}(\mathbf{x})) = \mathbf{f}(\mathbf{x})$$



Regularization with Dropout

- At test time, we replace the masks by their expectation
  - This is simply the constant vector 0.5 if dropout probability is 0.5
  - For single hidden layer: equivalent to taking the geometric average of all neural networks, with all possible binary masks
- Can be combined with unsupervised pre-training
- Beats regular backpropagation on many datasets
- Ensemble: Can be viewed as a geometric average of exponential number of networks.