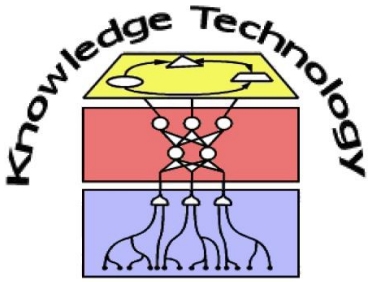
Neural Networks

Lecture 10: Training Neural Networks



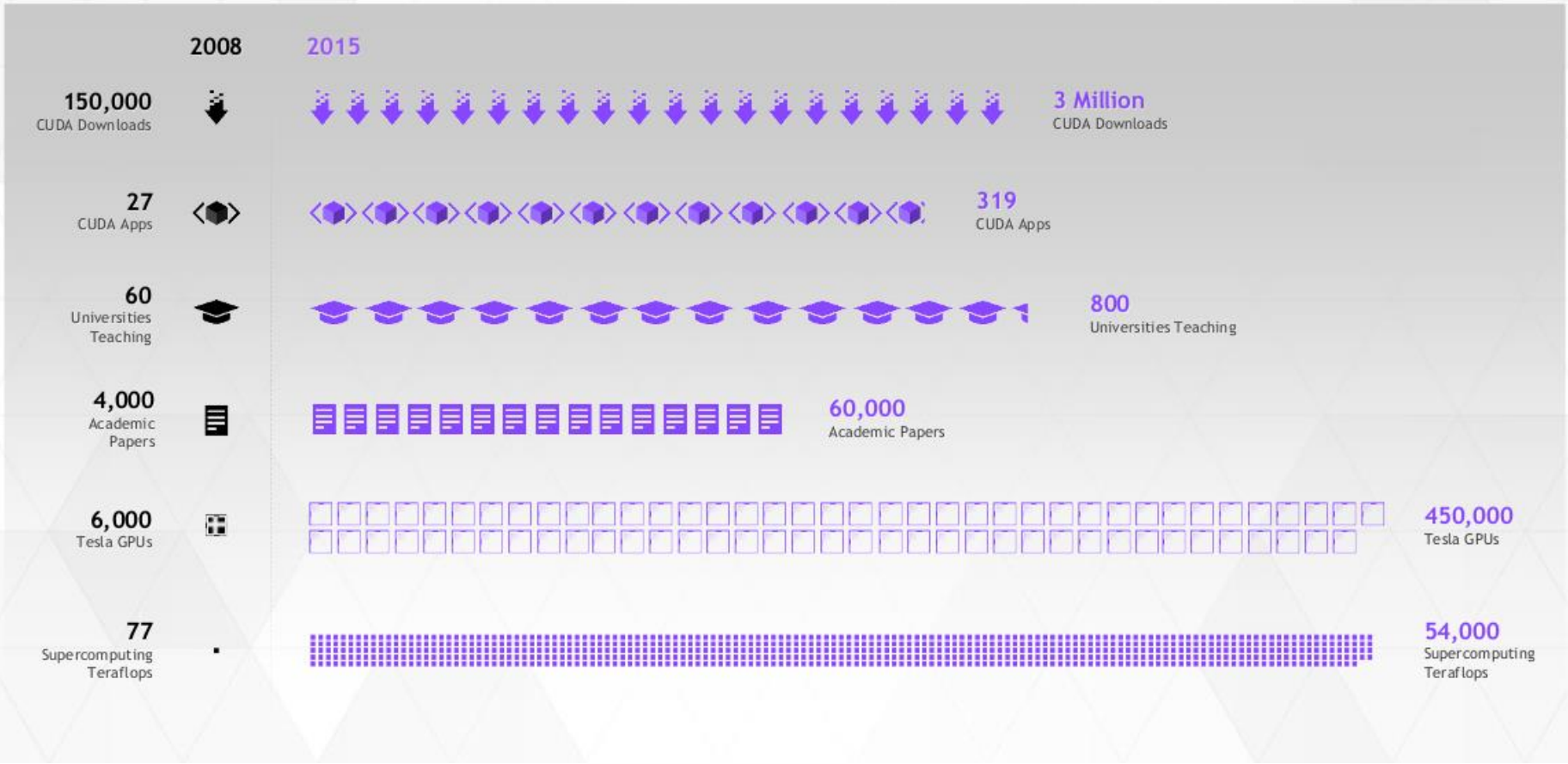
http://www.informatik.uni-hamburg.de/WTM/

Motivation

Deep Learning in practice:

- Implementing a neural network
  - GPU Computing and Computational Graphs
  - Symbolic programming
  - Deep Learning Frameworks
- Best practices for successful training
  - Combat overfitting
  - Hyperparameter Optimization
- MNIST example in Tensorflow and Keras

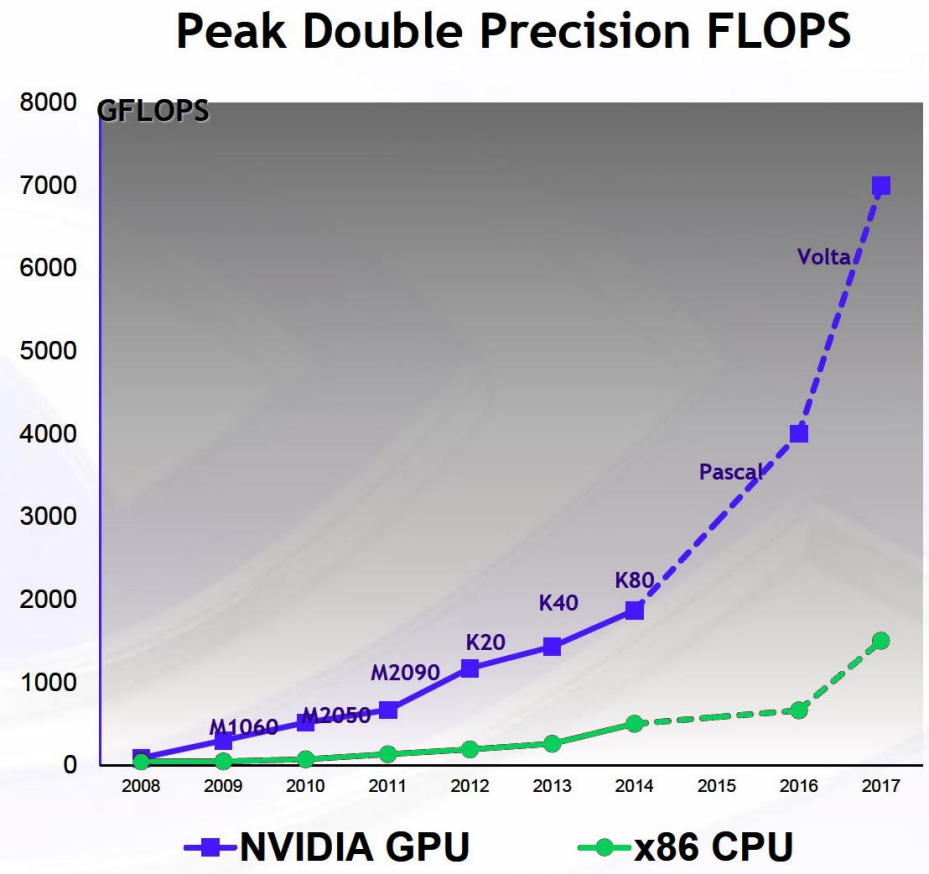
GPU Computing 2008 vs 2015

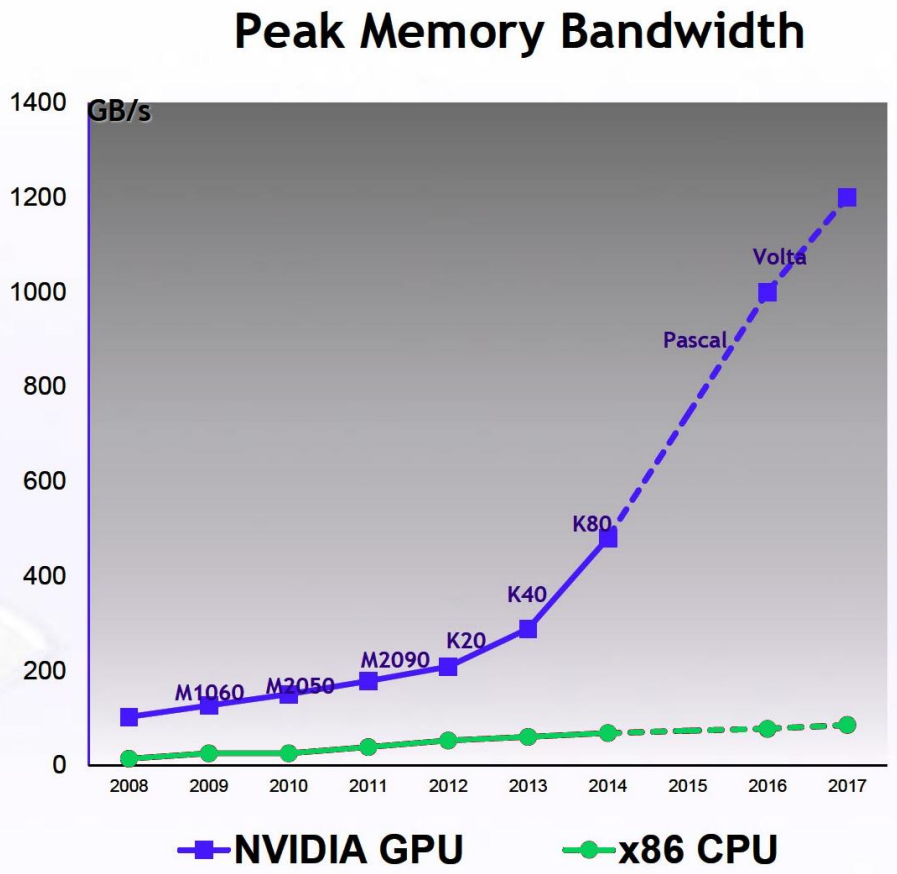


10x growth in GPU computing

[NVIDIA presentation at GPU Technology Conference 2015]

GPU vs CPU

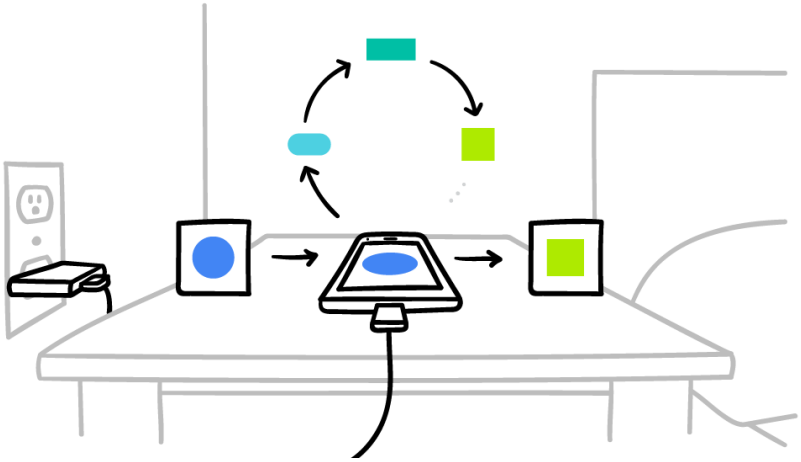




[NVIDIA presentation at GPU Technology Conference 2015]

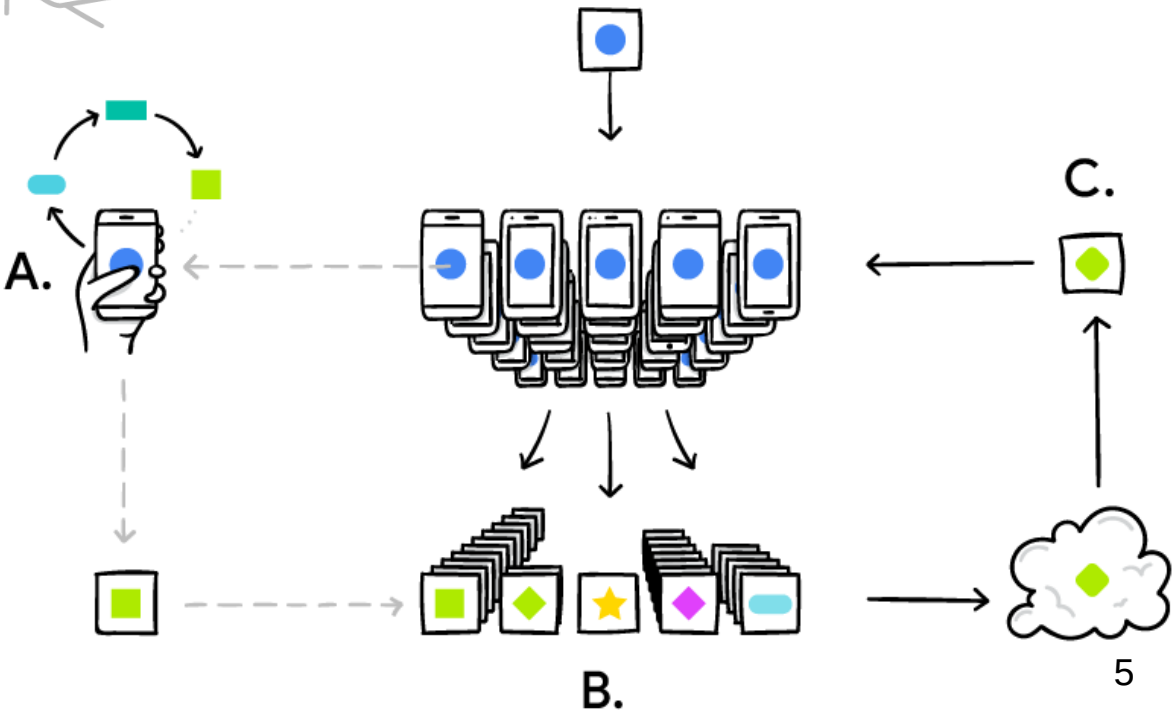
Massively Distributed Gradient Descent

"Federated Learning" (Google 2017)



- **A:** Phone personalizes model locally, based on usage
- **B:** Aggregate of many users' updates
- **C:** Consensus change to shared model

- Phone participates when idle



Batches in Gradient Descent

3 possibilities on *when* to update the weights:

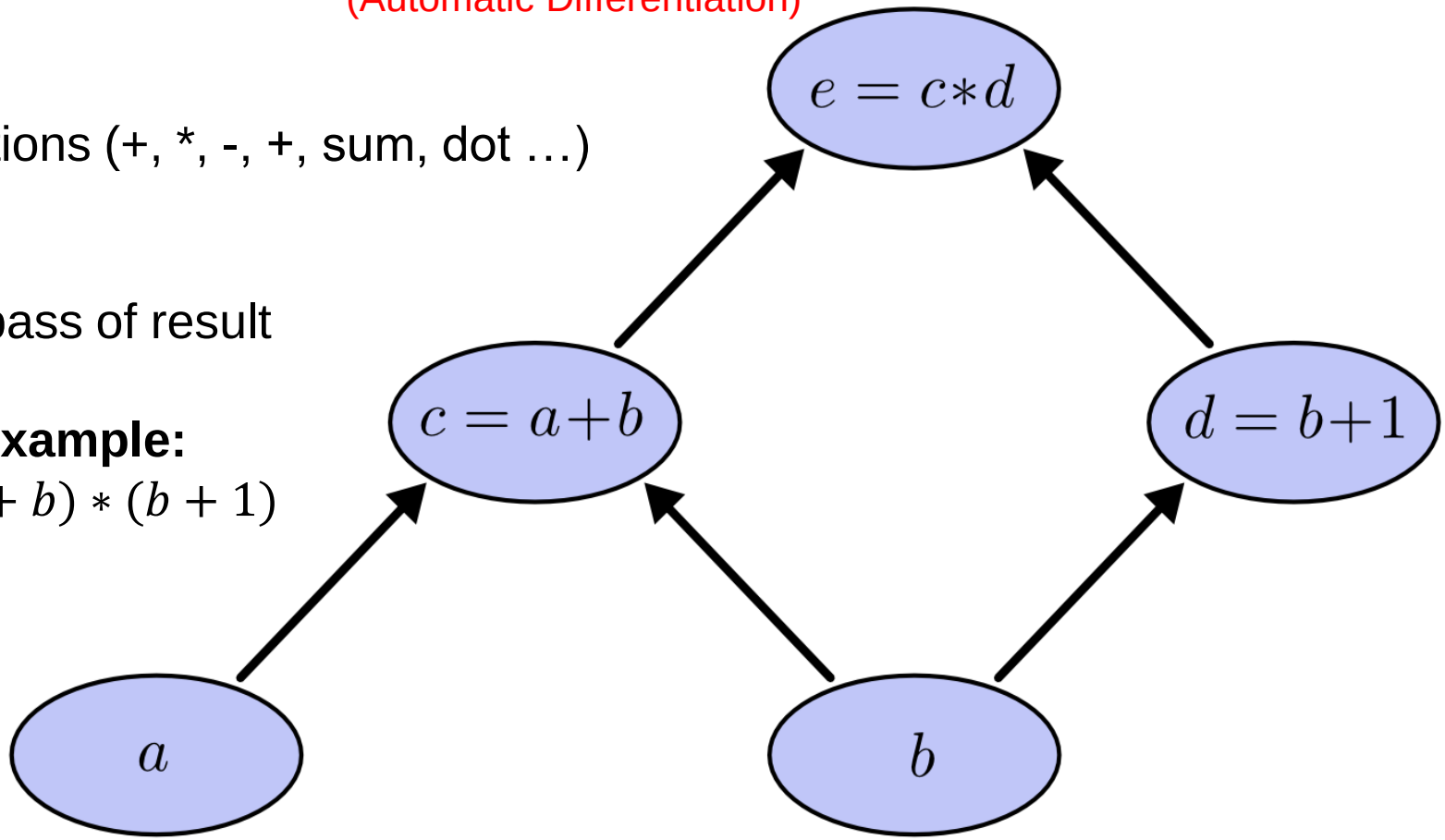
- Stochastic gradient descent (incremental/online training):
  - After each training sample
  - batch_size = 1
- Batch gradient descent (batch training):
  - After seeing all training samples
  - batch_size = $|D_{tr}|$
- Mini-batch gradient descent (mini-batch training):
  - After seeing a subset of the training data
  - 1 < batch_size < $|D_{tr}|$

In GPU: We use mini-batches, split them into smaller mini-batches, run them in parallel.
Generally: Compute gradient parallel for each sample in batch, then sum + average.

Computational Graphs

(Automatic Differentiation)

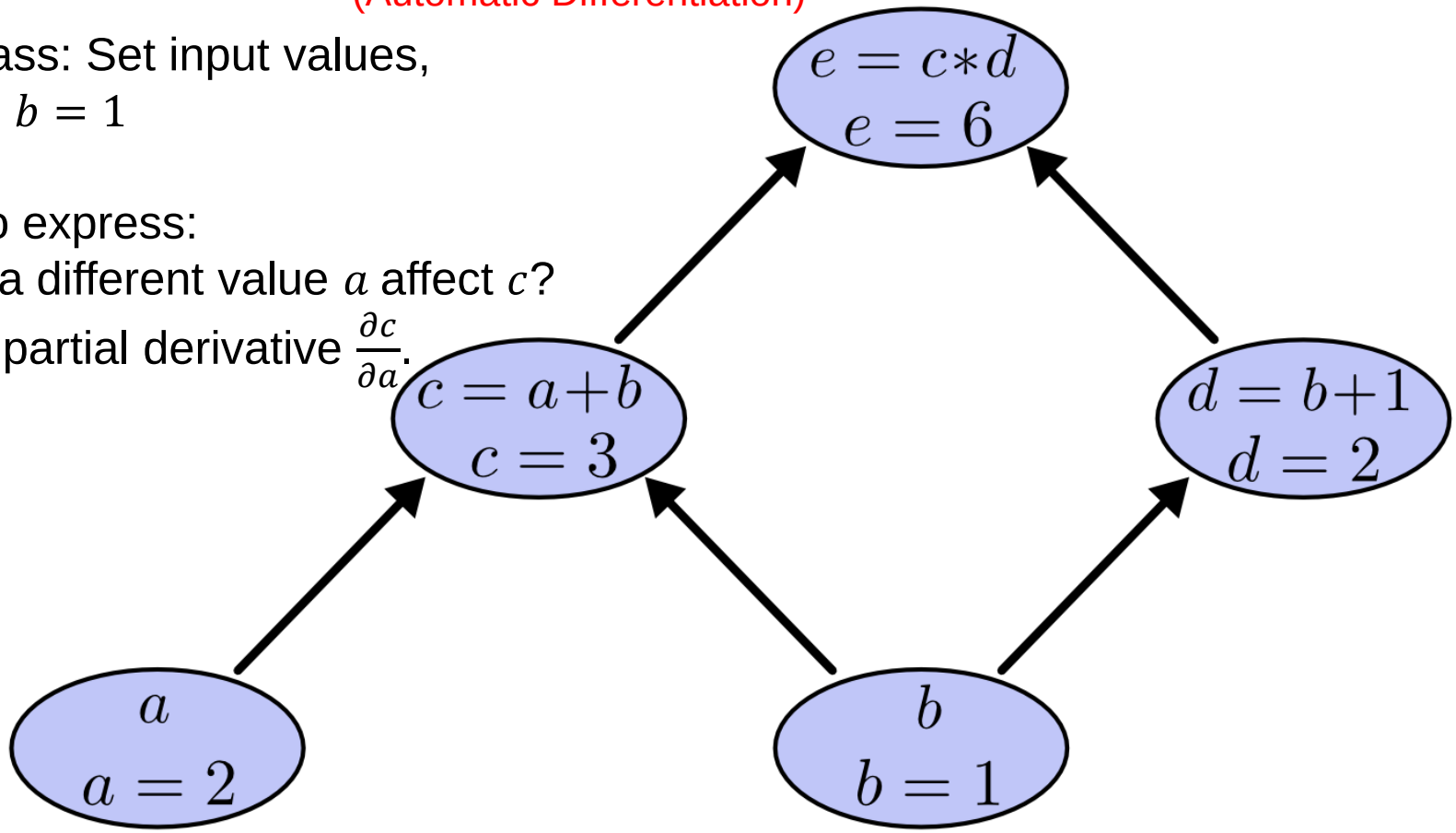
- **Nodes:**
  computations (+, *, -, +, sum, dot …)
- **Edges:**
  forward-pass of result
- **Shown example:**
  $e = (a + b) * (b + 1)$



Computational Graphs

(Automatic Differentiation)

- Forward pass: Set input values, e.g. $a = 2,\ b = 1$
- We want to express: How does a different value a affect c?
- This is the partial derivative $\frac{\partial c}{\partial a}$.



Partial derivatives:

With sum and product rule:

$$\frac{\partial c}{\partial a} = \frac{\partial}{\partial a}(a + b) = \frac{\partial a}{\partial a} + \frac{\partial b}{\partial a} = 1$$

$$\frac{\partial}{\partial u} uv = \mathrm{u}\frac{\partial v}{\partial u} + v\frac{\partial u}{\partial u} = v$$

But how do inputs a and b affect e?

Computational Graphs

(Automatic Differentiation)

Partial derivatives:

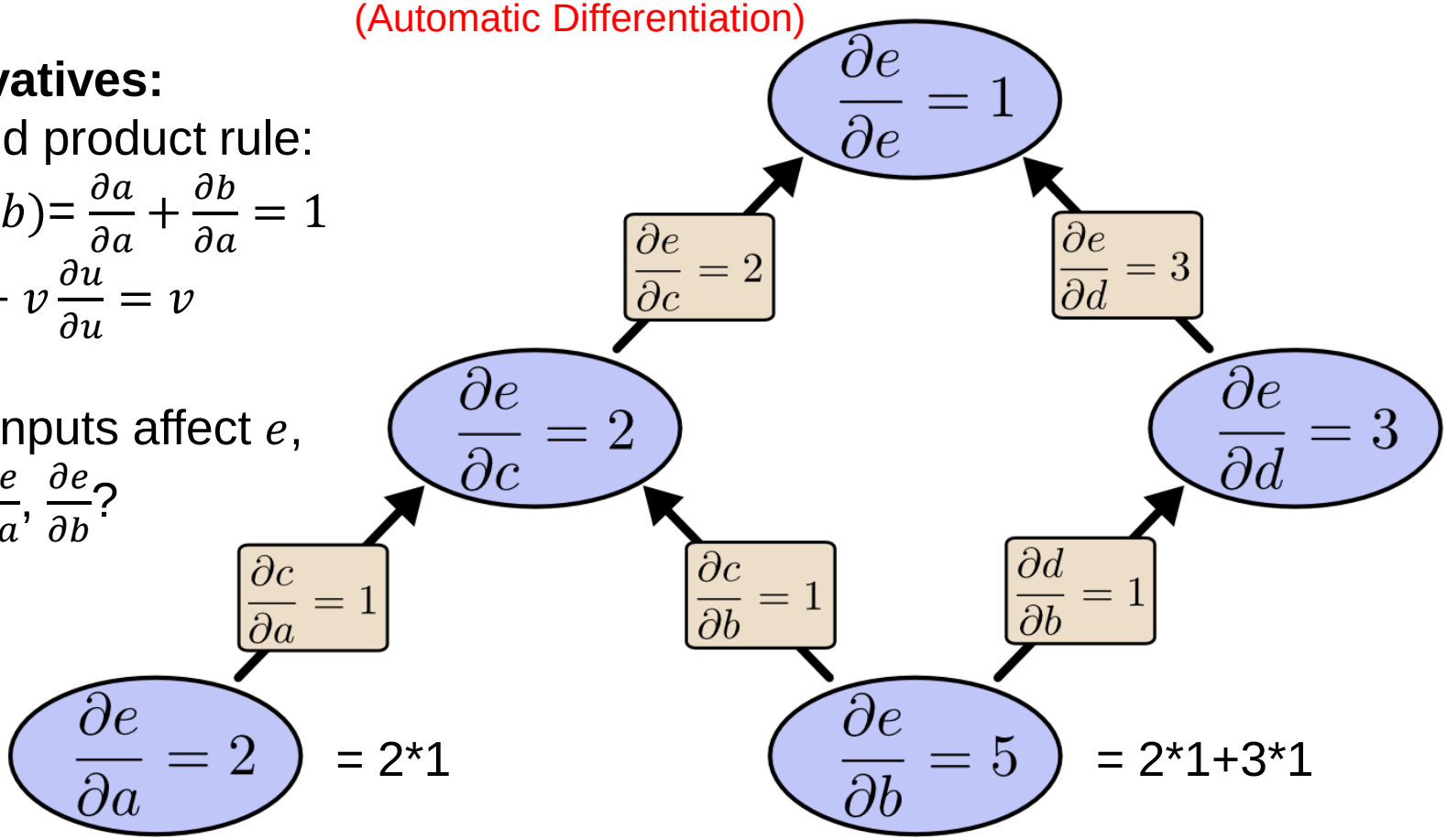
With sum and product rule:

$\frac{\partial c}{\partial a} = \frac{\partial}{\partial a}(a+b) = \frac{\partial a}{\partial a} + \frac{\partial b}{\partial a} = 1$

$\frac{\partial}{\partial u} uv = \mathrm{u}\frac{\partial v}{\partial u} + v\frac{\partial u}{\partial u} = v$

But how do inputs affect e, i.e. what is $\frac{\partial e}{\partial a}$, $\frac{\partial e}{\partial b}$?

Chain rule!

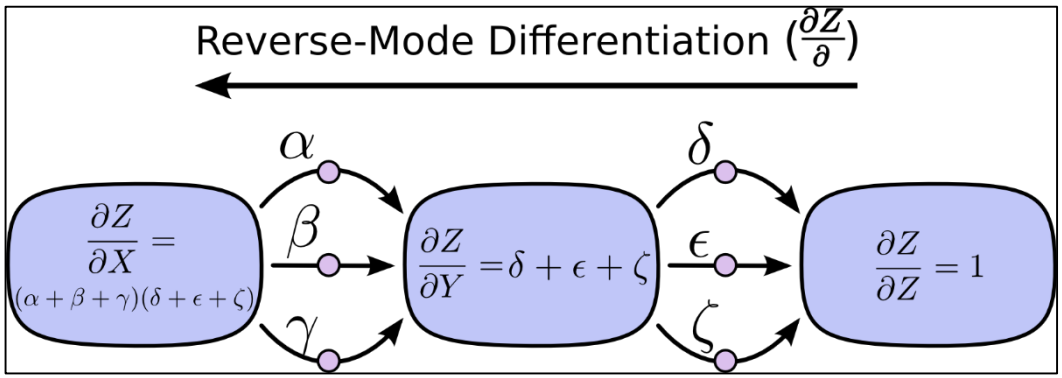


= 2*1

= 2*1+3*1

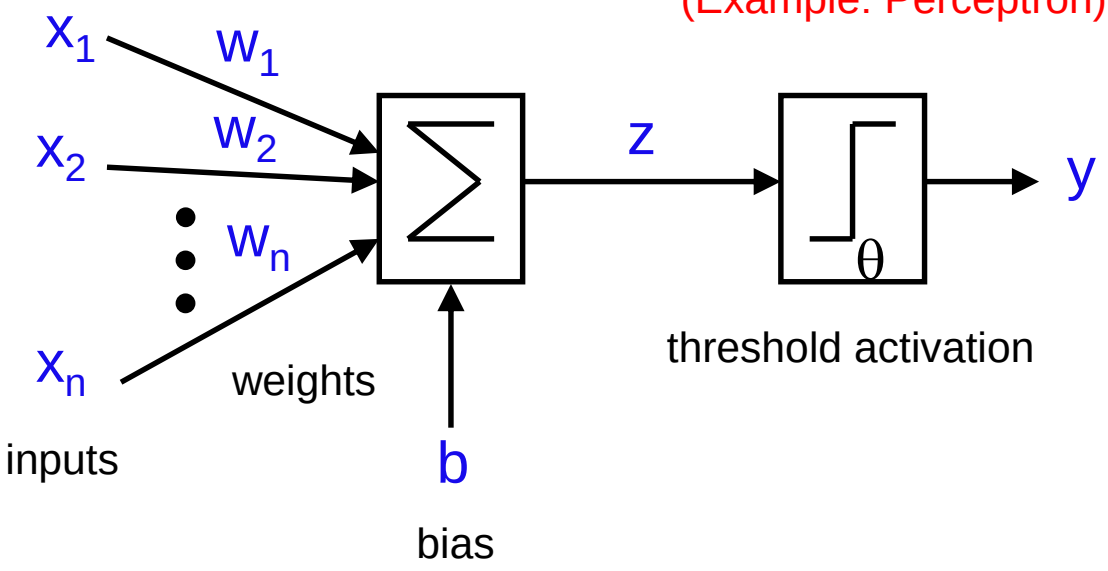
Reverse-Mode Differentiation:

Factor the paths backwards starting at e. Gives derivative of e with respect to ***every*** node!



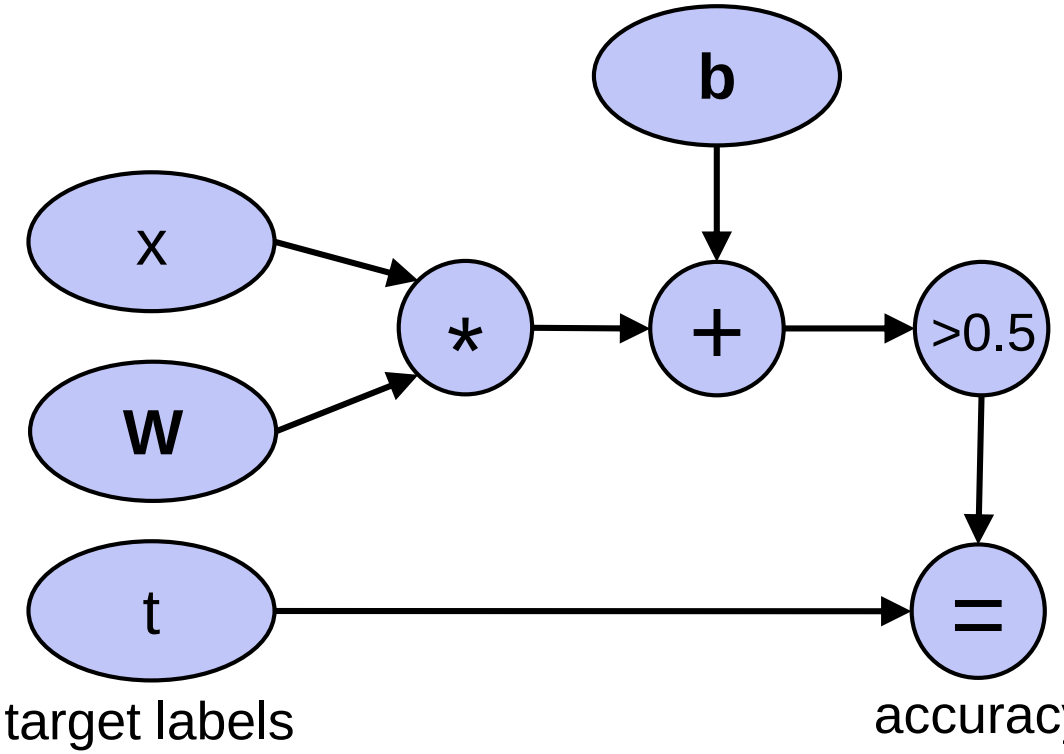
Computational Graphs

(Example: Perceptron)



$$z = \sum_i x_i w_i + b$$

$$y = \begin{cases} 1 \text{ if } z \geq \theta \\ 0 \text{ otherwise} \end{cases}$$



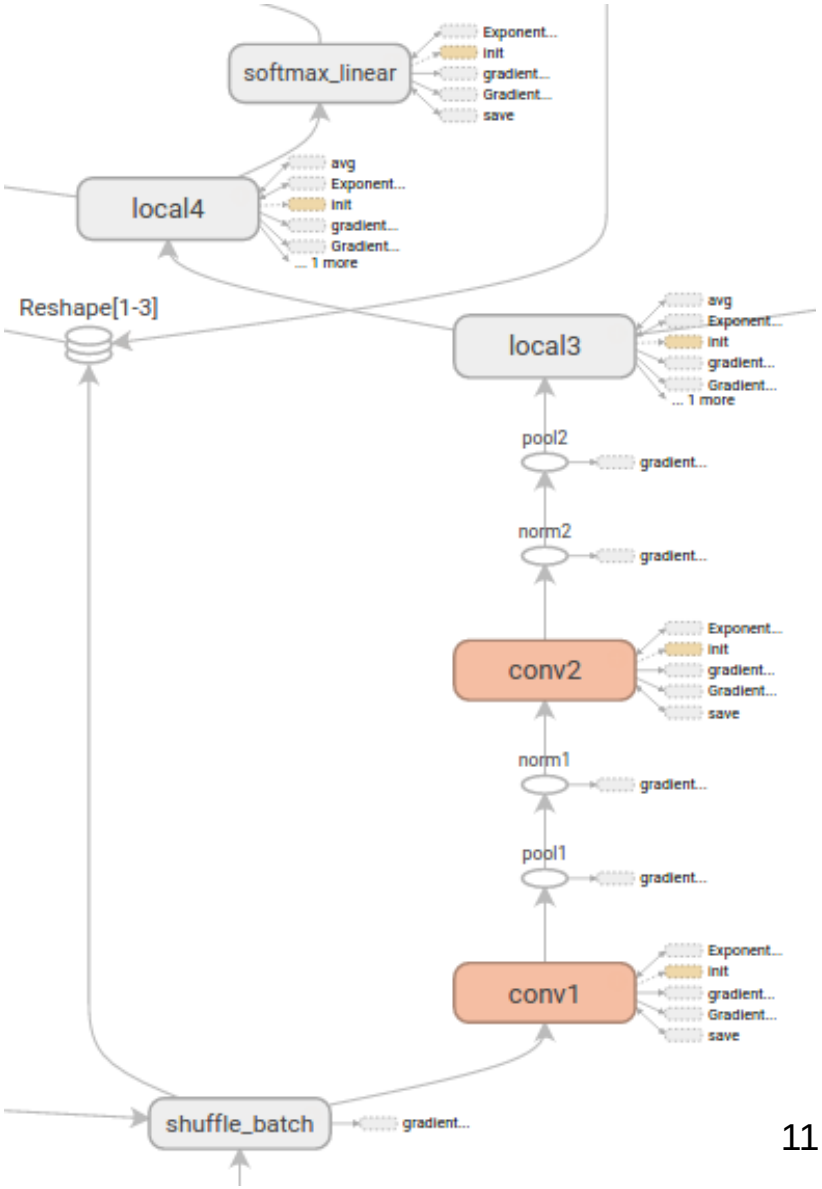
Computational Graphs

Why use computational graphs?

- Gradients „for free“
- Easy parallelization: Computation naturally segmented
- Computational model (computational graph) very close to conceptual model (neural network)

Problem:

- Symbolic programming requires rethinking



Deep Learning Frameworks

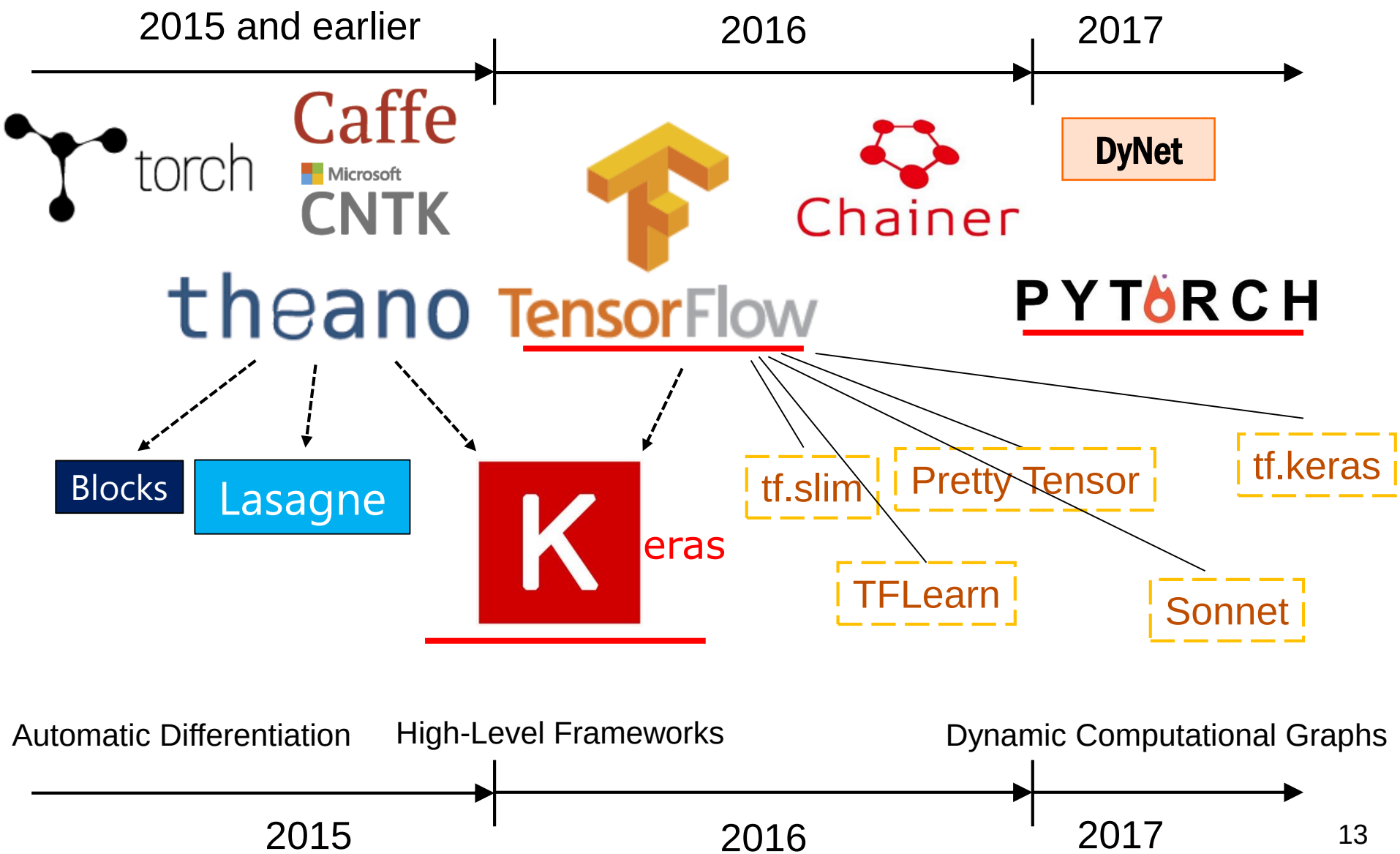
Advantages:

- Easily build big computational graphs
- Automatically compute all gradients
- Run it efficiently on GPU

Apache Singa, Azure, BidMach, Brainstorm, Caffe, Caffe 2[1], CNTK, Chainer, Deeplearning4j, Dlib, Dynet, Kaldi, Keras*, Leaf, MatConvNet, MaxDNN, minpy, MXNet[3], Neural Designer, OpenNN, Paddle, PyTorch[1], Torch[1], TensorFlow[2] (TF-Slim[2*], TFLearn[2*], Pretty Tensor[2], Sonnet[4]), Theano (Lasagne, Blocks, rllab)

[1]:Facebook, [2]:Google, [3]:Amazon/MIT/CMU/UWA,
[4]:DeepMind, [*]: Ships with TensorFlow (May 2017)

Current Framework Landscape



Comparison: Keras and Tensorflow

General:

- Tensorflow offers both high- and low-level abstraction
- Keras is a high-level wrapper for Tensorflow (and Theano, CNTK, MXnet)
  - Abstraction: Define models layer-by-layer

Documentation:

- Keras has many diverse examples
- Tensorflow has some beginner tutorials

Speed:

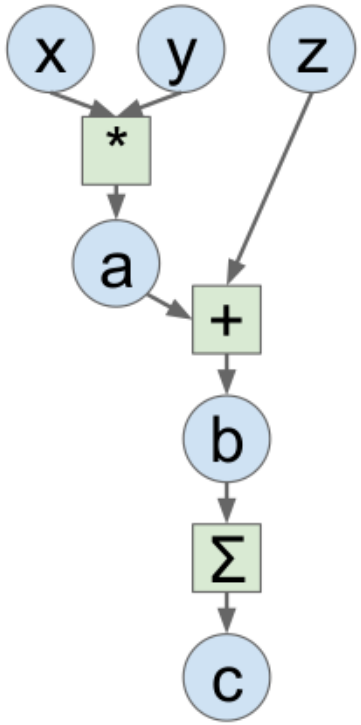
- No actually reliable (up-to-date) benchmarks exist
- The differences are usually not massive (only in some ”advanced” models)

Debugging:

- Everything symbolic: can’t use traditional debuggers
- Tensorflow offers Tensorboard
- Keras debugging difficult (due to abstraction)

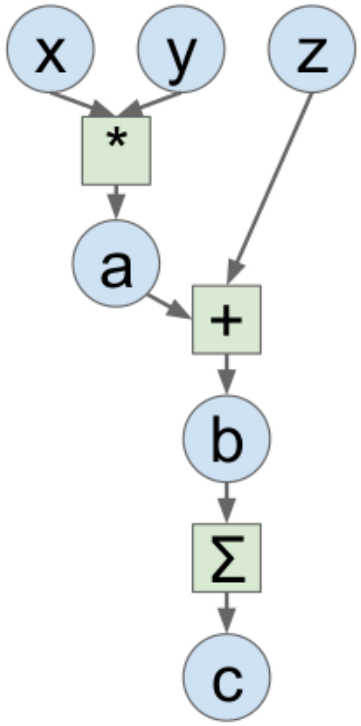
Numpy

```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)
```



Numpy

```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```

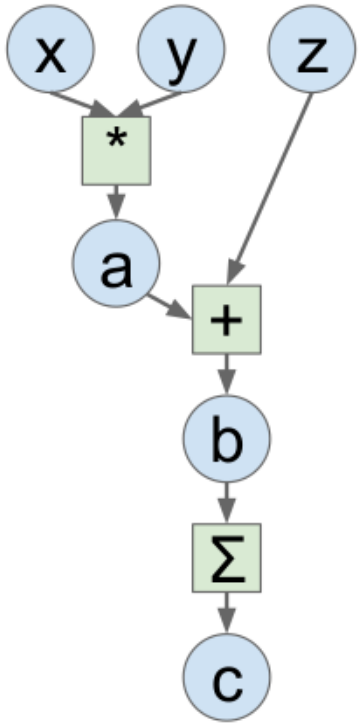


Drawbacks:

- Have to compute gradients yourself
- Can’t run directly on GPU

Numpy

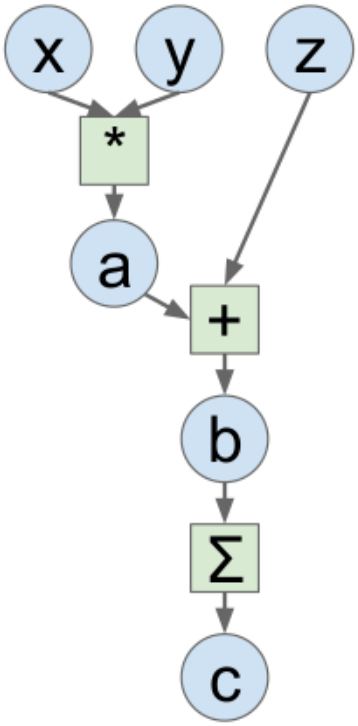
```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```



TensorFlow

```
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4

with tf.device('/gpu:0'):
    x = tf.placeholder(tf.float32)
    y = tf.placeholder(tf.float32)
    z = tf.placeholder(tf.float32)

    a = x * y
    b = a + z
    c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                   feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

TensorFlow



run on GPU →

create graph

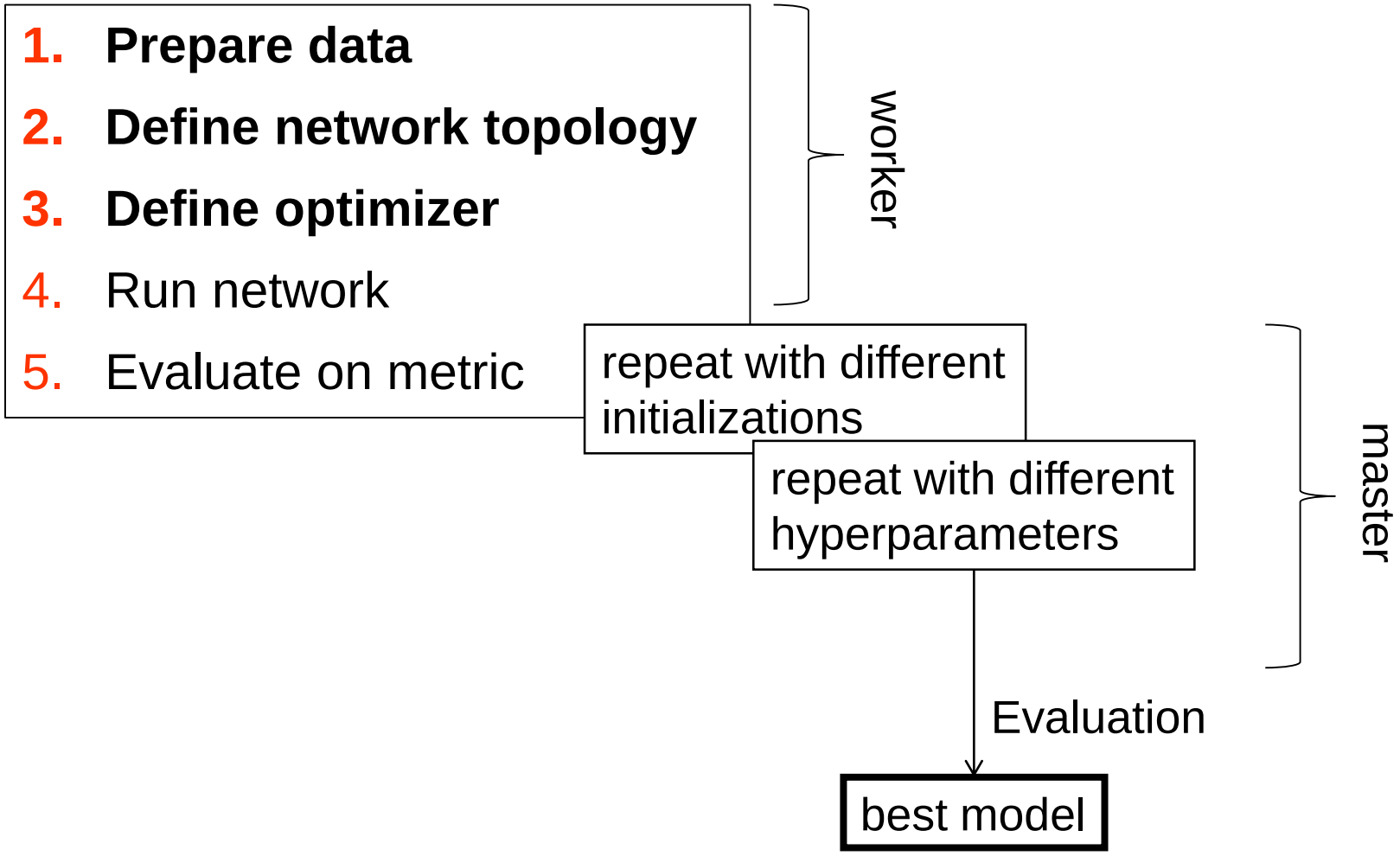
tell TensorFlow to compute gradients →

initialize Session with actual values, run the computation

```
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4

with tf.device('/gpu:0'):
    x = tf.placeholder(tf.float32)
    y = tf.placeholder(tf.float32)
    z = tf.placeholder(tf.float32)

    a = x * y
    b = a + z
    c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                   feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

tf.placeholder: tensor-object that can hold a value during runtime
tf.Session: environment in which operations are executed (nothing is actually computed outside of it!)

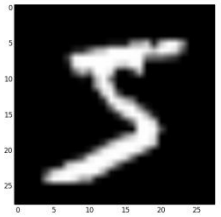
Main parts of any neural network implementation



- Workers can be distributed in parallel on different instances

Example: MNIST with 1-layer NN

- MNIST: Dataset for classification of handwritten digits
- We start with a single-layer network



```
# input X: 28x28 grayscale images, the first dimension (None)
will index the images in the mini-batch
X = tf.placeholder(tf.float32, [None, 28, 28, 1])
# correct answers will go here
Y_ = tf.placeholder(tf.float32, [None, 10])
# weights W[784, 10]   784=28*28
W = tf.Variable(tf.zeros([784, 10]))
# biases b[10]
b = tf.Variable(tf.zeros([10]))
# flatten the images into a single line of pixels
# -1 in the shape definition means "the only possible
dimension that will preserve the number of elements"
XX = tf.reshape(X, [-1, 784])
Y = tf.nn.softmax(tf.matmul(XX, W) + b) # the model
```

[https://github.com/martin-gorner/tensorflow-mnist-tutorial/blob/master/mnist_1.0_softmax.py]

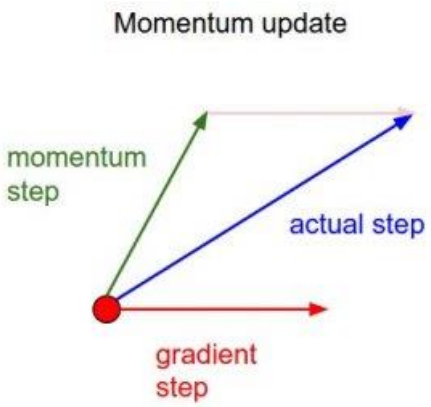
Example: MNIST with 1-layer NN

```
# log takes the log of each element, * multiplies the tensors element by element
# reduce_mean will add all the components in the tensor
# so here we end up with the total cross-entropy for all images in the batch
cross_entropy = -tf.reduce_mean(Y_ * tf.log(Y)) * 1000.0
# normalized for batches of 100 images,
# *10 because  "mean" included an unwanted division by 10


# accuracy of the trained model, between 0 (worst) and 1 (best)
correct_prediction = tf.equal(tf.argmax(Y, 1), tf.argmax(Y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
# training, learning rate = 0.005
train_step = tf.train.GradientDescentOptimizer(0.005).minimize(cross_entropy)

init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)
```

Yields 92% accuracy after 4 epochs – can we do better?

Optimizers

```
# vanilla update:
x += -learning_rate * dx

# momentum update:
v = mu * v - learning_rate * dx
x += v
```

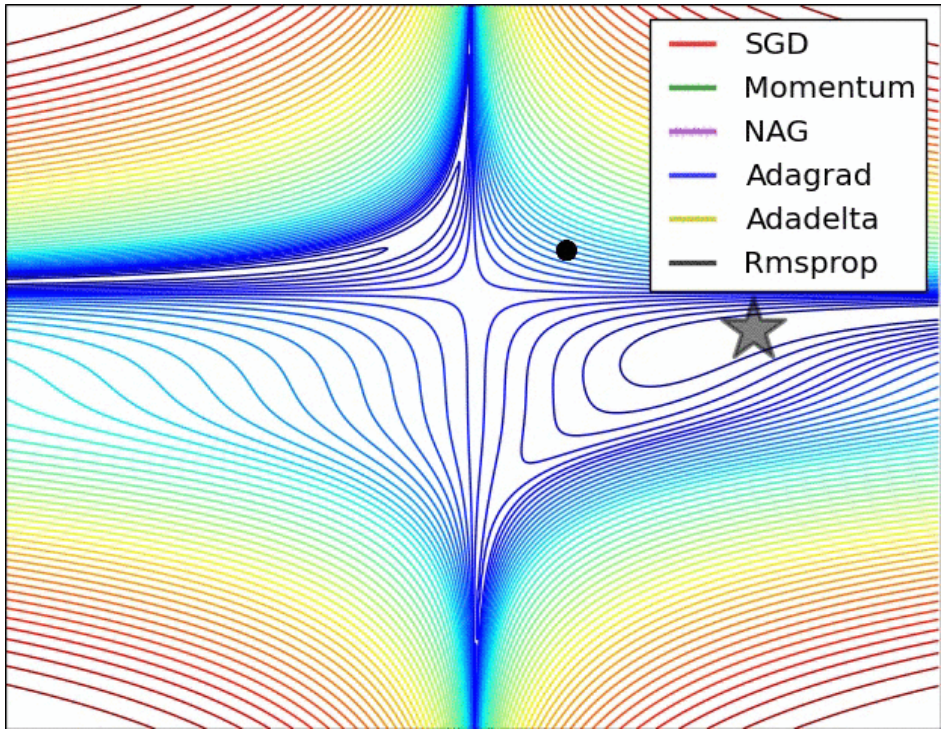


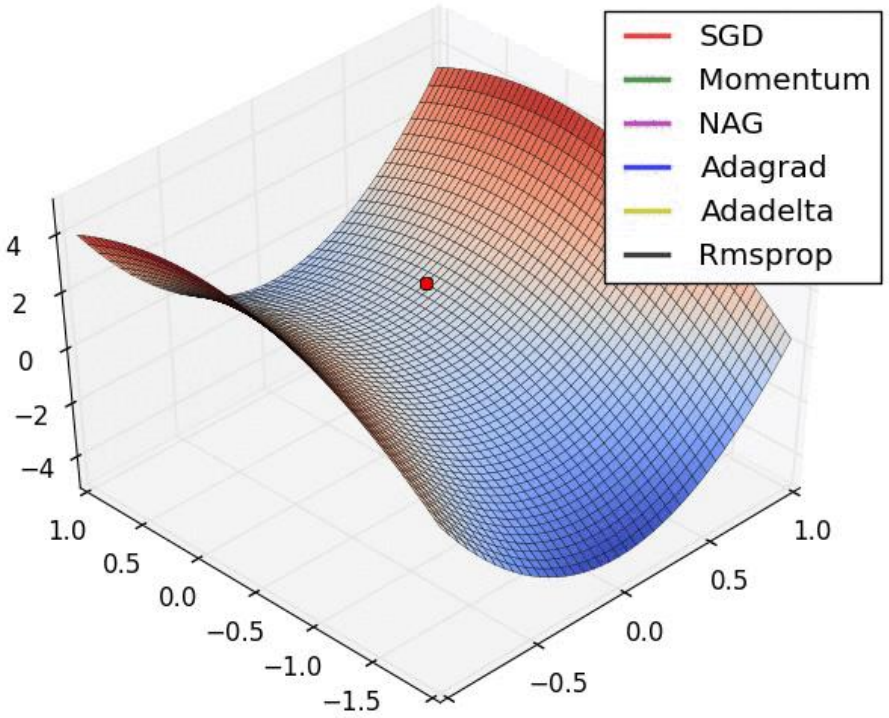
Optimizers with adaptive learning rates for each parameter:

- **Adagrad:** Maintains per-parameter learning rate.
  Good with sparse gradients.
- **RMSprop:** Like Adagrad but exponentially decaying mean of grad.
  Good with noisy gradients.
- **Adam:** Combines both by looking at the decaying mean and also variance of the gradients.
- All of these use a running average of past gradients

In practice: Adam or SGD with global but decaying learning rate (annealing) and momentum

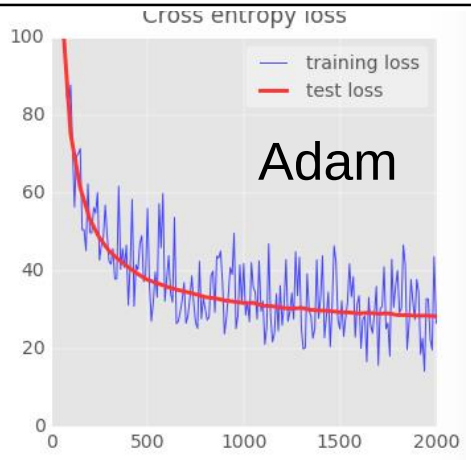
Optimizers

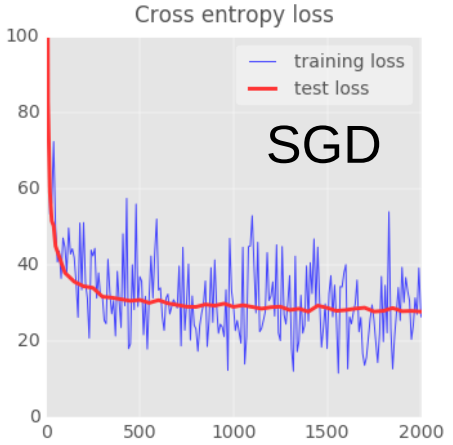





```
train_step = tf.train.AdamOptimizer().minimize(cross_entropy)
```

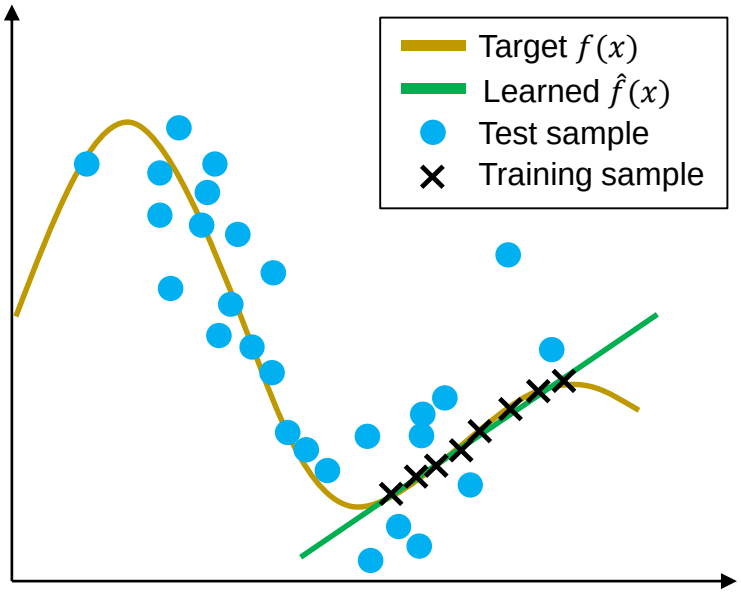
- Using Adam yields same loss as before but with smoother convergence
- But can we trust the accuracy on test data?





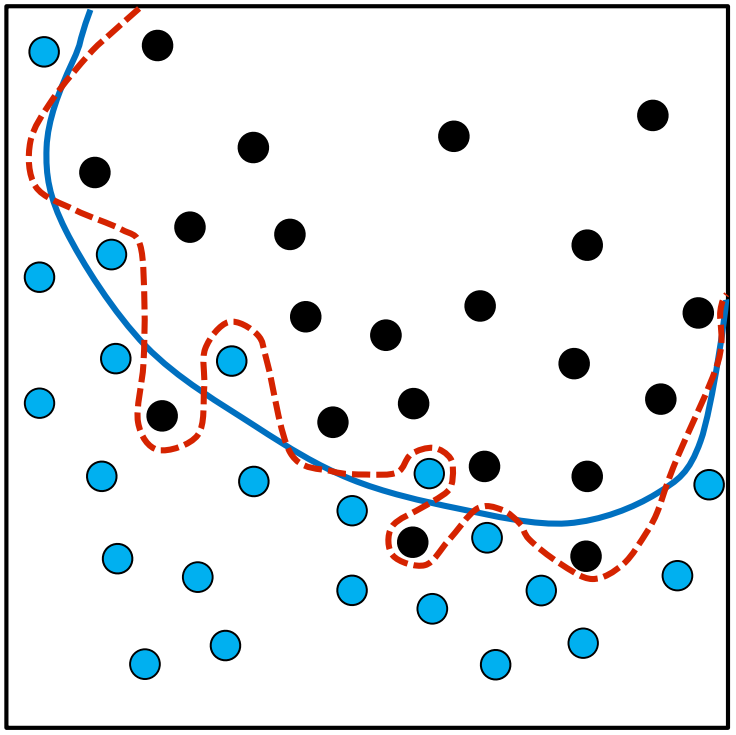
Picking test data

- Basic assumption: Training/test data from same distribution
- Covariate shift:
  - Training and test input distributions are different but target function remains unchanged → weak extrapolation



- Possible solution: Assign weights to data based on “importance”, rebalance distributions in training and test data

Overfitting



- We can get to 0% training error by memorization
- Memorization ≠ Generalisation
- All noise is captured, no learning
- Typically occurs when:
  - Training too long
  - Too few data
  - Too many parameters

- Potential Solutions:
  - More training data, data augmentation
  - Reducing architecture complexity (parameters)
  - Cross Validation, early stopping
  - Regularization

Note: These are only best-practices, not hard guarantees!

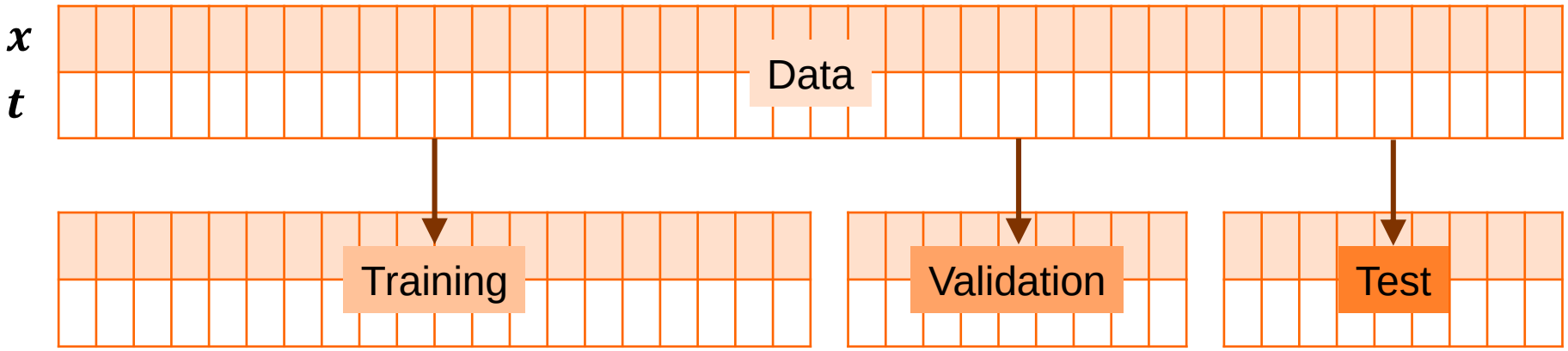
Cross Validation

To avoid overfitting we should use three different sets:

- Training set:
  - To train the model (weights with backprop)
- Validation set:
  - To find the right hyperparameters (by observation)
  - So the model doesn‘t overfit weights on the training set
- Test set:
  - To test generalization error on best chosen model
  - So *you* don‘t overfit hyperparams on the validation set

- It’s even possible for a community to methodically overfit:
  e.g. best accuracy on MNIST in 2013: **0.979**
  → *Any* data collection merely *models* the ground truth

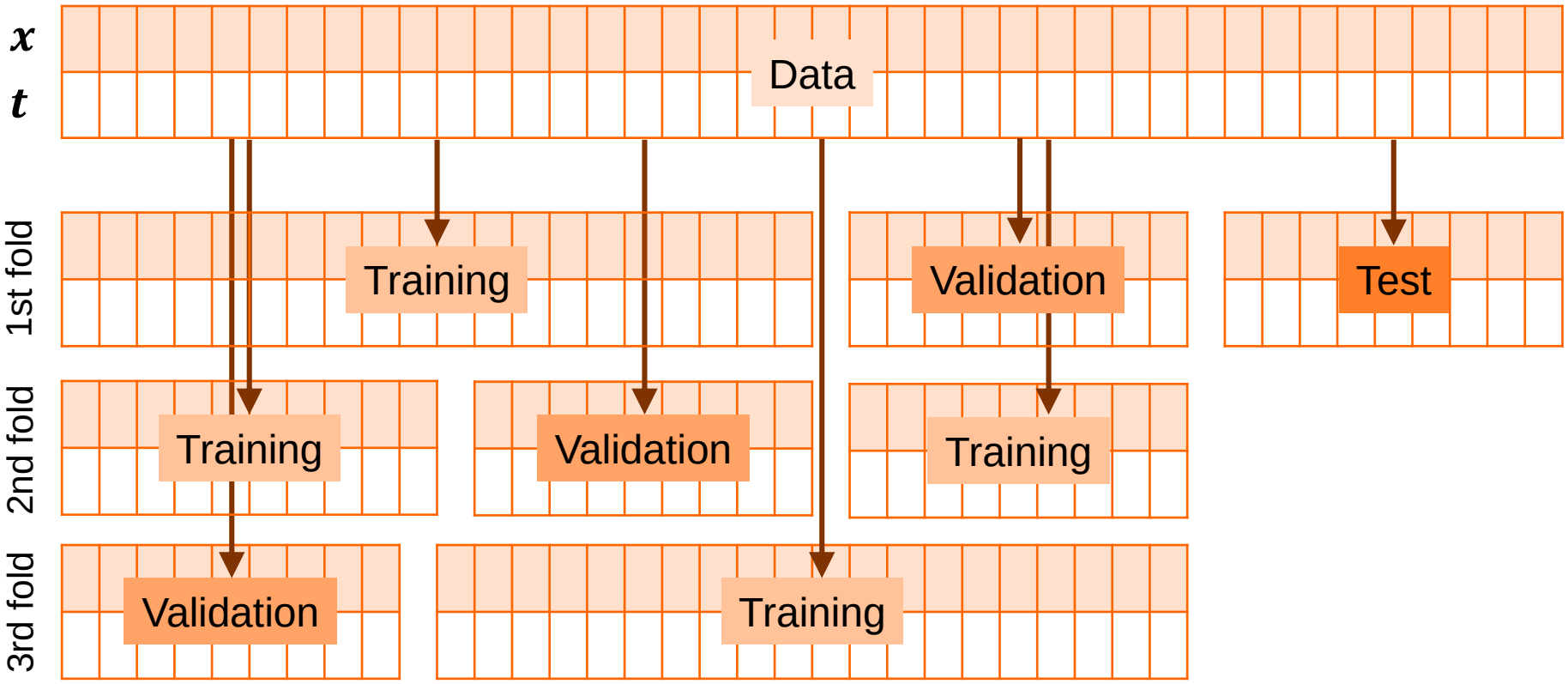
Holdout Cross Validation



- Independent sets. Watch variance + label distributions!
- Careful sampling is a prerequisite for CV to work:
  If there isn't enough data, it isn't helpful.
- Common splits:
  - 50%-25%-25%
  - 80%-10%-10% (Pareto Principle)

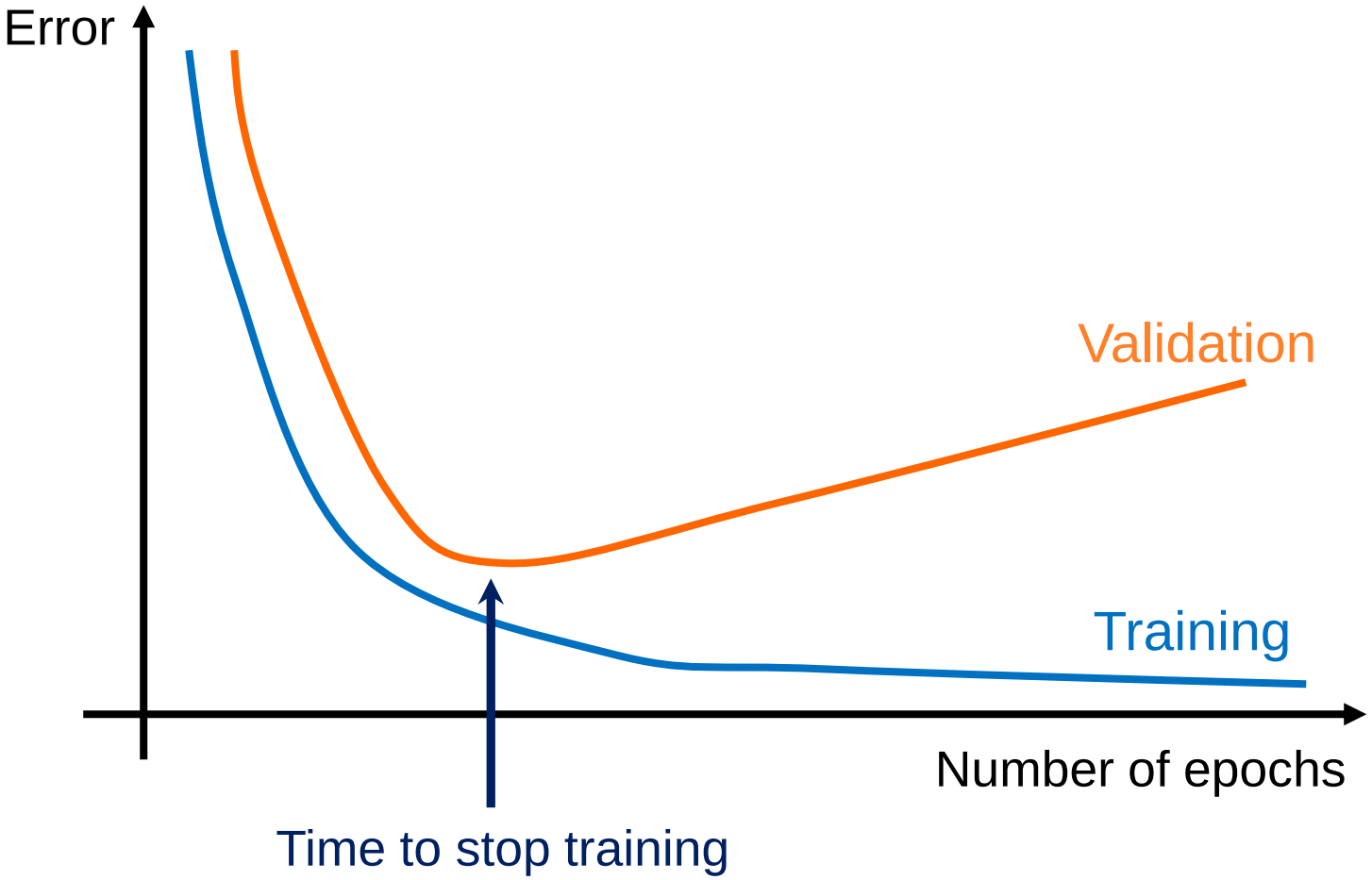
K-Fold Cross Validation

(Example: k=3)



- Resample validation data iteratively for each fold
- Larger k = less bias, more variance
- All data used for training and testing but requires k runs (average error).

Early Stopping



In practice: Stop if $\text{validation\_error} < \text{min\_improvement}$ during last $patience\ p$ epochs
Save a copy of the best model in the last p steps and choose it.

Regularization

- Goal: Tuning model complexity to prevent overfitting
- Main idea: Penalize large weights, forcing e.g. smoothness.
- Regularization term $\boldsymbol{R}(\boldsymbol{W})$ added to loss function (λ: regulariz. strength):

$$L(f(x), t) + \boldsymbol{\lambda}\, \boldsymbol{R}(\boldsymbol{W})$$

- **L1 Regularization:** $R(f) = \lambda \sum_k \sum_l |W_{k,l}|$
  - Leads to sparse weight vectors (many zero), ignoring noisy inputs
  - Has built-in feature selection (each $w_{k,l} = 0$ discards an input x_k)
- **L2 Regularization:** $R(f) = \lambda \sum_k \sum_l W_{k,l}^2$ ("weight decay")
  - Penalizes "peaky" weight vectors (many close to zero)
  - Optimum if all $w_{k,l} = 0$ but that leads to bad loss (R and L antagonistic)

```
regularizer = tf.nn.l2_loss(weights) # computes sum(x ** 2) / 2
loss = tf.reduce_mean(loss + lambda * regularizer)
optimizer = tf.train.AdamOptimizer().minimize(loss)
```

In practice: L2 with global cross-validated λ (start around 0.01).

L2 Regularization: Example

Given:

$x = [1, 1, 1, 1]$

$w_1 = [1, 0, 0, 0]$

$w_2 = [0.25, 0.25, 0.25, 0.25]$

Linear loss: $w_1^T x = w_2^T x = 1$

- L1: $R(f) = \lambda \sum_k \sum_l |W_{k,l}|$
- L2: $R(f) = \lambda \sum_k \sum_l W_{k,l}^2$

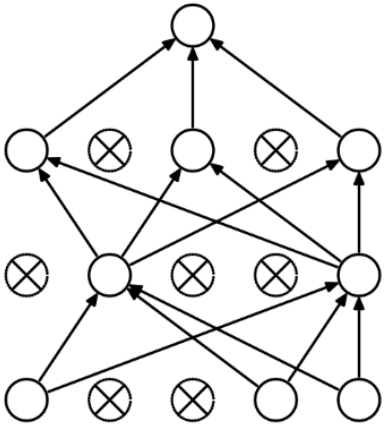
- L2 regularization:
  - Favors w_2 and „vectors with most non-zero entries“
  - Wants to spread out the information over all dimensions
  - This works because we then use more features
  - It does however prevent specialization of neurons (like dropout)
  - Specialization is a nice property (visualisation!) but can lead to complex co-adaptations between neurons (overfitting)

Demo: Regularization

- Go to http://playground.tensorflow.org
- Make a large neural net
  (many layers, units, use all input features)
- Use the Gaussian dataset with noise=50
- Compare L1 vs L2 regularization with different rates (0.003-0.3) until you stop overfitting
- Hover over the weights to understand the results, especially the L1 feature selection

Deep Learning techniques

- **Dropout:** (Regularization)
  - Mostly for deep neural networks (CNNs)
  - During training each neuron has probability p of being active or being set to zero otherwise
  - Can only be used during training



- **Batch Normalization:**
  Normalize activations to zero mean and unit variance
  - Solution to *"internal covariate shift":*
    - All input distributions will change drastically over time.
    - Influenced by: parameter initialization, learning rate, activation function
  - Advantages:
    - Reduced int. cov. shift
    - Reduced dependence of gradients on weight init and scale
    - Allows nonlinearities, higher learning rates

- $p = 0.3$ ... 0.5 for Dropout in CNNs.
- BatchNorm after fully connected or convolutional layers (but before nonlinearity)

Hyperparameter Optimization

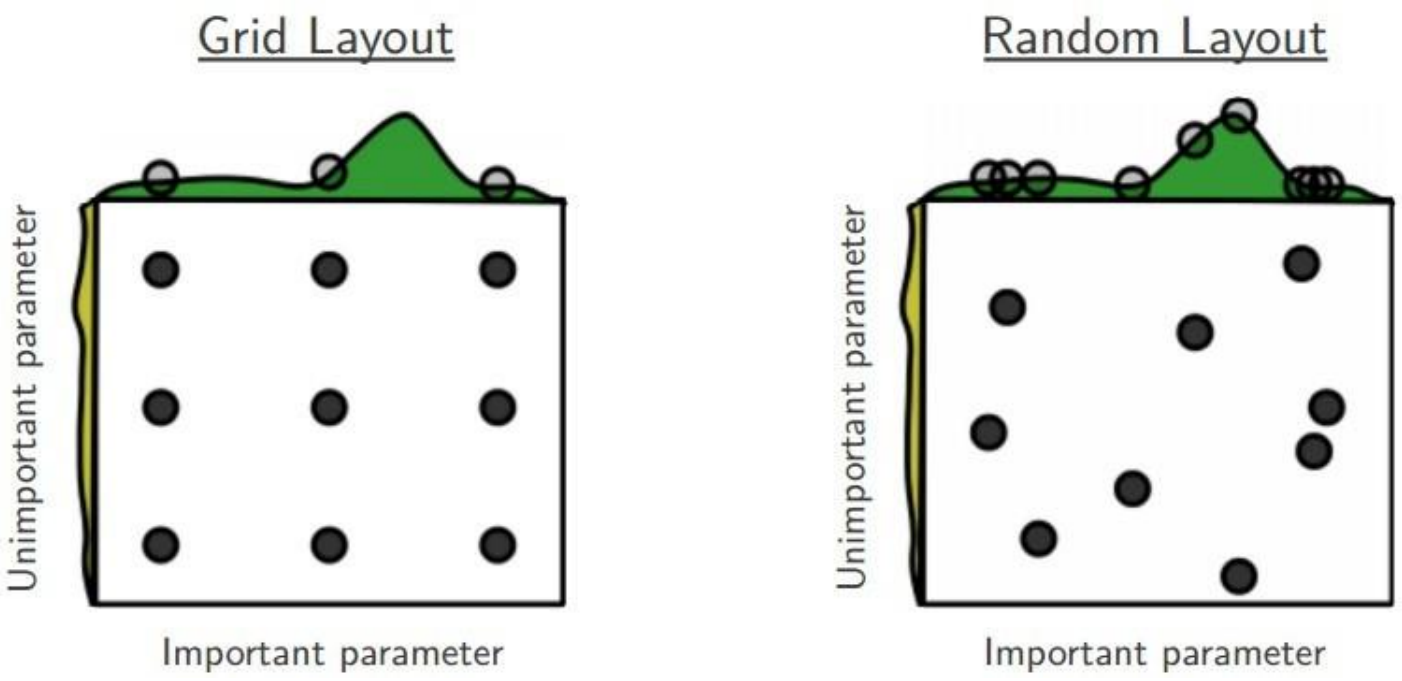
Why being an expert takes time and practice

- With introduced complexity comes an increase in hyperparameters:
  - regularization, learning rate, hidden units, number of layers, convolution (stride/filters), epochs, momentum
- We optimize these by monitoring the validation loss
- Systematic procedures:
  - Intuition + Grid Search (most common)
  - Random Search
  - Bayesian Optimization (best in theory)
  - Evolutionary Algorithms, Gradient-based, ...

Try out heuristics from literature but never fully rely on them!

In practice: Intuition first. Then informed Random Search or Bayesian Opt. with *hyperopt*. Start with a coarse search, then run it for more epochs and initializations.

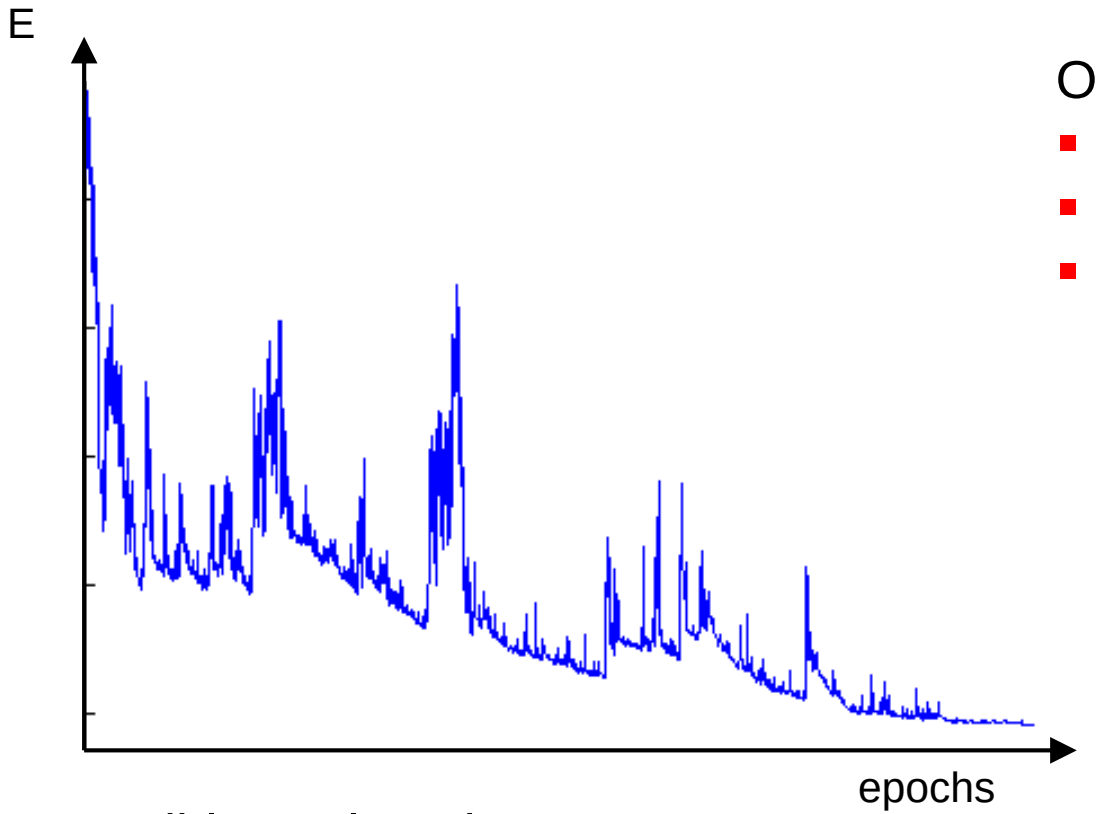
Grid Search vs Random Search



Random seach better than naive grid search [Bergstra12]
(not all hyperparameters are significant)

- Asynchronous and stoppable/pausable at any given time
- But non-adaptive: Still beaten by decisions of a real expert
- Bayesian approaches are more intelligent
  (but hard to parallelize & have own hyperparameters)

Quiz: How can we improve convergence?



Observations:

- "Wobbly", oscillations
- Spikes somewhat periodic
- Converges but each wrong decision takes a short time to correct

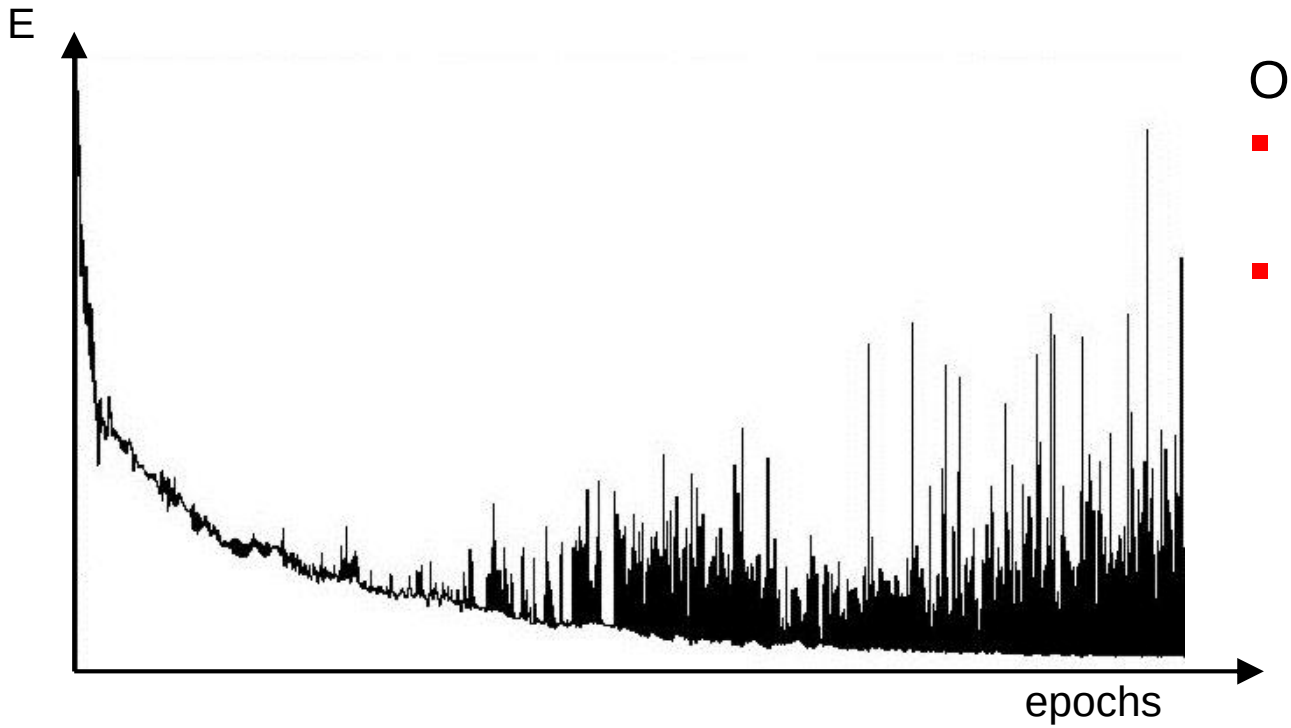
Possible explanation:

- Online learning with unshuffled training data, possibly imbalanced

Things to try:

- Shuffle training samples, vary batch_size, lower learning rate (decay)

Quiz: How can we improve convergence?



Observations:

- Increasing oscillations over time
- Still converges

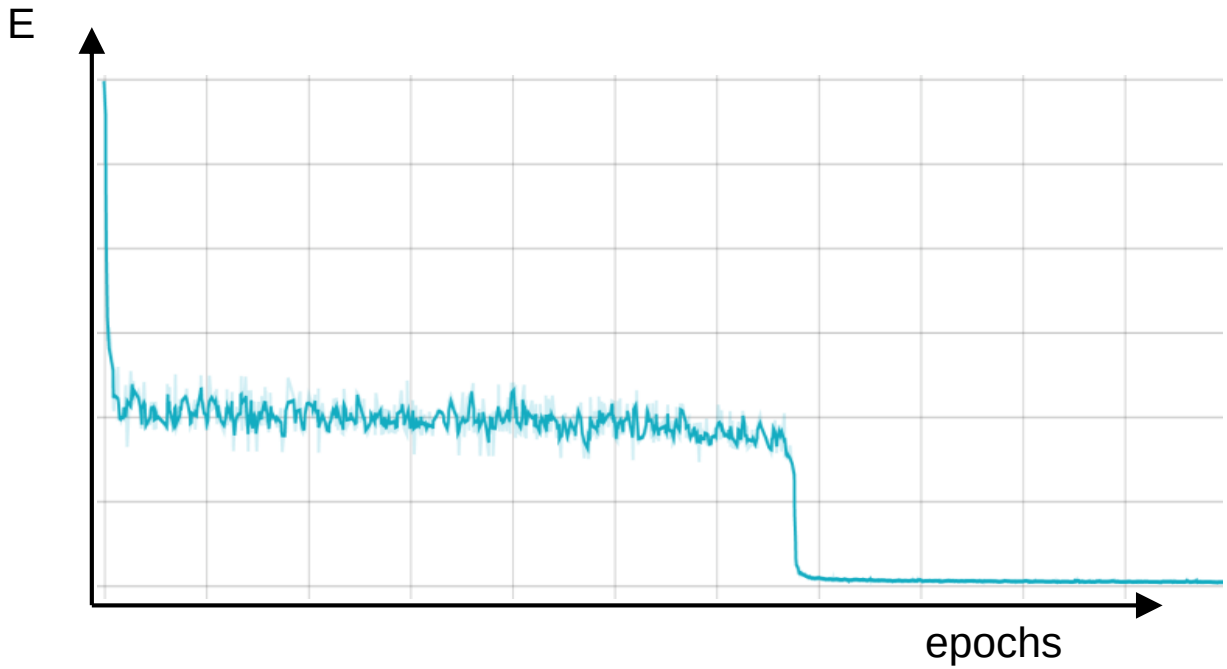
Possible explanation:

- Stuck between local minima, numerical instability through large gradients

Things to try:

- Shuffle training samples, try online learning,
- Batch Normalization, activation function (range)
- Lower learning rate (decay)

Quiz: How can we improve convergence?



Observations:

- Plateau for a long time
- Sudden drop in error

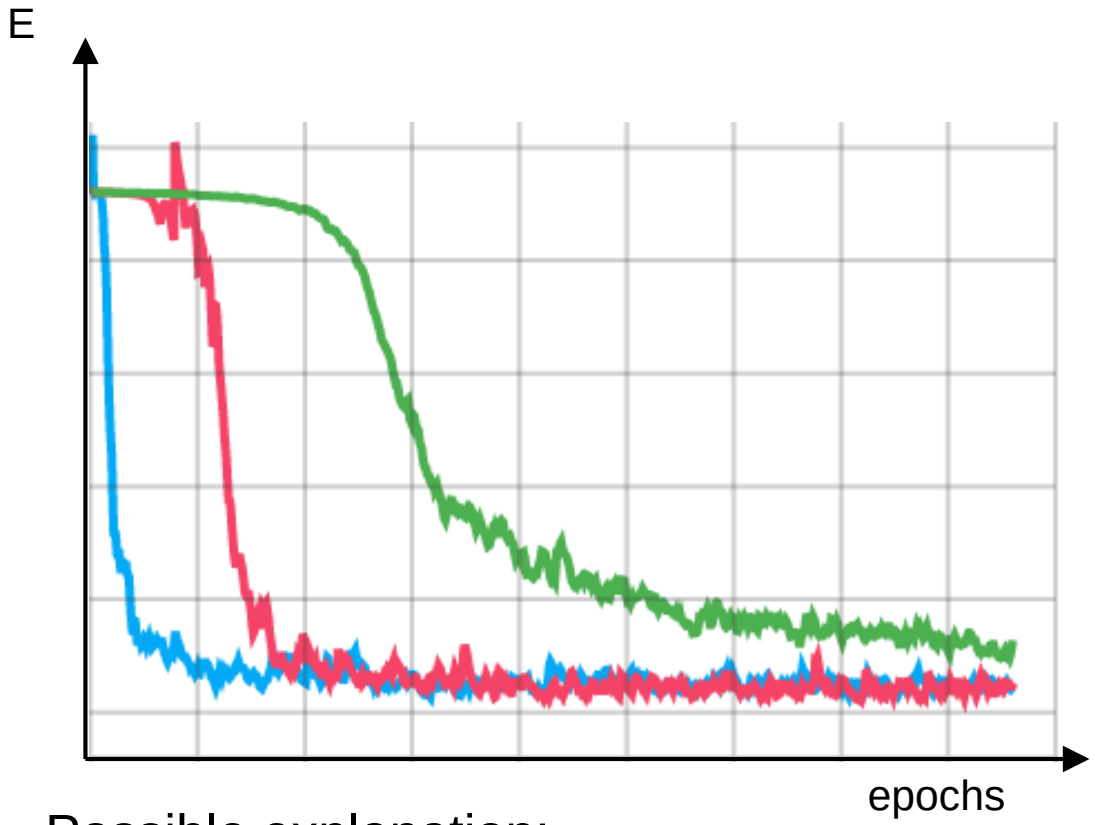
Possible explanation:

- Stuck in local minimum early on
- Overfitting near end?

Things to try:

- Decaying learning rate (needs to be higher at start)
- Test different wheigth initializations

Quiz: How can we improve convergence?



Observations:

- 3 runs with different initializations
- Converges well over time but initial performance very different

Possible explanation:

- Bad initialization (plateaus at start)

Things to try:

- More robust initialization schema
- Input normalization, batch normalization
- Check slightly larger learning rate

Debugging Neural Networks

- Most frameworks are symbolic with static graphs:
  We can‘t actually debug the networks during training!
- We can plot the Computation Graph („Tensorboard“)

- Check edge cases for hyperparameters
- Log everything and start data mining (visualizations!)
- If all else fails: pen + pencil

- Most importantly: Make sure you **start simple** and gradually increase complexity. Otherwise...
  - Reduce number of techniques and „tricks“
  - Simplify pre-/postprocessing methods
  - Simplify network architecture and complexity

Simplifying our implementation for MNIST (1)

We will use Keras for convenience :

- Import preimplemented layers and optimizers
- Define Hyperparameters
- Prepare data:
  - Reshape 2D images tor 1D-layer input
  - Normalize to [0,1]
  - Make 1-hot vectors for target labels

```
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.optimizers import Adam

batch_size = 128;
num_classes = 10;
epochs = 20

# the data, shuffled and split between train and
test sets
(x_train, y_train), (x_test, y_test) =
          = mnist.load_data()
x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255

# convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train,
          num_classes)
y_test = keras.utils.to_categorical(y_test,
          num_classes)
```

Simplifying our implementation for MNIST (2)

We will use Keras for convenience :

- Define our model layer by layer
- Compile instruction: loss, optimizer, metric
- model.fit: Run a session and train the network
- Evaluate (98.40% accuracy after 20 epochs)

```
model = Sequential()
model.add(Dense(512, activation='relu',
        input_shape=(784,)))
model.add(Dropout(0.2))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(10, activation='softmax'))
model.summary()

model.compile(loss='categorical_crossentropy',
        optimizer=Adam(),
        metrics=['accuracy'])

history = model.fit(x_train, y_train,
        batch_size=batch_size,
        epochs=epochs, verbose=1,
        validation_data=(x_test, y_test))

score = model.evaluate(x_test,
                       y_test,
                       verbose=0)

print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

Improving our implementation for MNIST

Using a convolutional neural network (CNN) instead of an MLP:

- Conv2D and MaxPooling2D: input can stay 2D, doesn't need reshape
- Flatten into 1D before fully connected layers
- Improves results from 98.40% to 99.25%+

```
(...)

model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
       activation='relu',
       input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, (3, 3),
       activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes,
       activation='softmax'))

(...)
```

Further improvements of our CNN

- Data augmentation
  - Rotate and shift images, zero-center normalization
- Weight initialization
  - Xavier initialization, bias to 0
- Batch Normalization
  - For all convolutional/fully connected layers
- Exponential decay of learning rate
- Using ensembles
- Proper hyperparameter optimization

Further Reading

- Andrew Ng: Machine Learning Yearning (unfinished)
- LeCun: Efficient Backprop
- Computational Graphs:

  http://colah.github.io/posts/2015-08-Backprop/
- Keras MNIST CNN example:

  https://github.com/fchollet/keras/blob/master/examples/mnist_cnn.py
- Tensorflow MNIST examples:

  https://www.tensorflow.org/get_started/
- Check CommSy for more material

Exams

Next week: Deep Reinforcement Learning

Exam Dates (register @ Studienbüro 27.06.-04.07.):

- Prof. Wermter:
  - **09.08.18**, **10.08.18**
  - **26.09.18**
- Dr. Weber:
  - **08.08.18, 10.08.18**
  - **27.09.18**

http://www.inf.uni-hamburg.de/studies/orga/dates/2018-suse-oral-exams.html