

# Accelerator-Based Programming - Neural Networks Tensorflow

Jörn Zimmerling

September 7, 2022



UPPSALA  
UNIVERSITET

# Deep learning

- Deep learning uses **artificial neural networks** and belongs to machine learning methods (representation learning)
- Artificial neural networks have existed since the 1960s
  - “Multi-layer perceptrons” were popular in the 1980s
- Deep neural network:
  - Artificial neural networks with multiple layers between input and output layers
  - Composition of primitives → features from lower layers can be composed to approximate more complicated relationships
  - Deep networks can model complex non-linear relationships
  - Components in network: neurons, synapses, biases, functions
  - Imitate functioning in human brain
  - Networks get **trained**
- Course focus on computational aspects than actual models
- Get some familiarity with relevant techniques



# Layers in a neural network

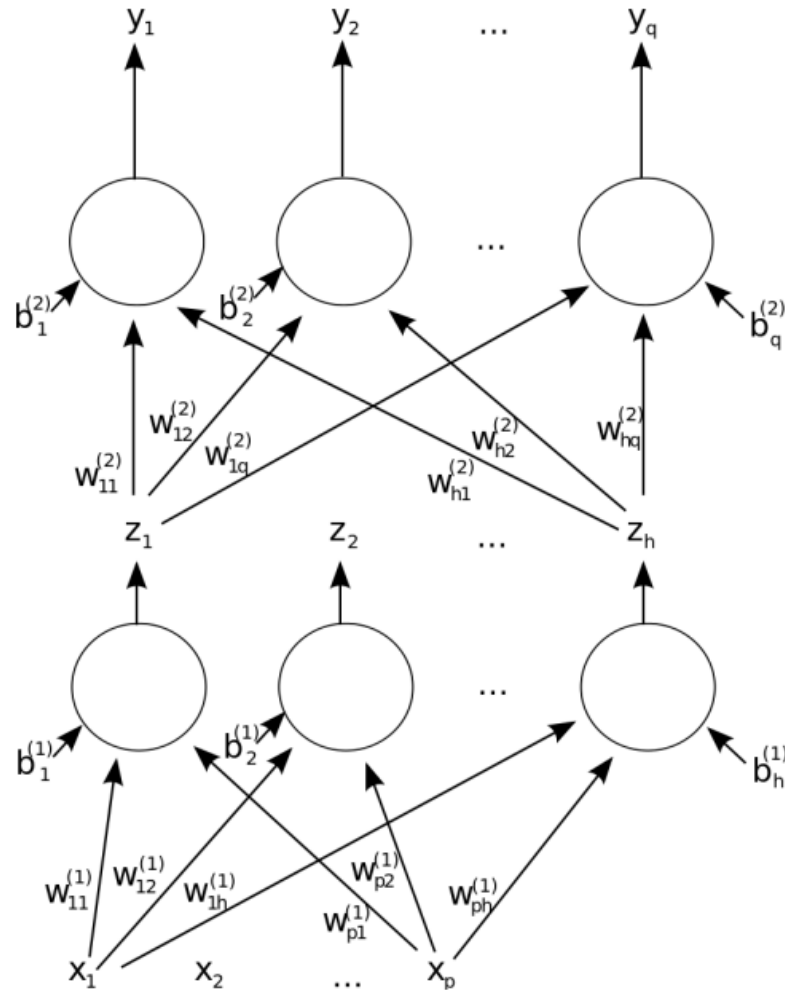


Image source:

<https://commons.wikimedia.org/w/index.php?curid=9516009>

- Simple two-layer network with inputs  $x$ , outputs  $y$
- Inputs are multiplied by weights
- Summation along all connections (synapses) to a node (neuron)
- Add bias to each sum
- Interpretation with multi-dimensional linear regression: Weights  $w_{ij}$  linear contribution, bias is intercept,

$$z(x) = w^T x + b$$

- Apply activation function
- Result from activation functions forms output, fed to the next layer



## Network as a graph

- The network is a directed (acyclic) graph
  - Called a feed-forward network
    - Data flows from input layer to output layer
    - No flow back
  - General, biological, neural circuits do not need to behave like this
- Edges (connections) indicate the flow of data



# Backpropagation

- Define desired outputs ( $y$  in our example)
- Define how to compare computed outputs against the desired outputs (objective function/loss function)
  - Simple example: mean square error
- How can we minimize the loss?
  - Adjust connection weights to compensate for each error
  - Divide error among connections from a node
  - Mathematical approach: Compute the gradient along the network
  - If we know the activation and the gradient for the output step, we can also compute the gradient for any variable and activation function backwards
- Take a small step in the direction of lower loss value for every single optimizable variable in the network
  - Some form of gradient descent





## Example: Gradient descent method

- Typical objective for backpropagation: Find  $\mathbf{w} \in \mathbb{R}^n$  such that

$$Q(\mathbf{w}) = \frac{1}{n} \sum_{i=1}^n Q_i(\mathbf{w})$$

is minimal

- Summand  $Q_i(\mathbf{w})$  typically associated with  $i$ -th observation of data set
- Standard gradient descent method:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \eta_k \nabla Q(\mathbf{w}_k) = \mathbf{w}_k - \frac{\eta_k}{n} \sum_{i=1}^n \nabla Q_i(\mathbf{w}_k)$$

- With  $\eta_k$  step size in iteration  $k$  (learning rate)
- Gradient evaluation seems cheap on small models, but
  - Needs to take into account all summand functions' gradients
  - $\sim$  quadratic complexity
- Simplification: Stochastic gradient descent
- Sample a subset of summand functions at every step
- Effective for large-scale data sets
- $\sim$  linear complexity



# Stochastic gradient descent and mini-batches

- Simplest way to overcome the need to store the activation value for every single node in the network, for every input is to approximate the true gradient by a gradient on a single instance  $i$ ,

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \frac{\eta_k}{n} \nabla Q_i(\mathbf{w}_k)$$

- Index  $i$  can be randomly chosen
- Compromise between work on a single index  $i$  and all indices is to compute gradient against multiple training examples, called **mini-batches** at each step
- Significantly better performance than single-instance stochastic gradient
- More samples in a batch might allow greater compute parallelism
  - Can use vectorization in evaluation and increase data locality
  - Especially important for GPUs with abundant arithmetic capabilities but limited caches
- But too many samples in a batch can exhaust memory and reduce cache performance



# Quality of stochastic gradient descent

- Just tracking the gradient seems simple
  - But it is extremely slow
    - Need small steps (low learning rate) to avoid overshooting
  - Objective function not convex
    - Moving towards the top of a hill won't bring us to top of Mount Everest
- Avoid getting trapped in local minima by different batch configurations and optimization algorithms
  - Use different random subsets for each batch such that the stable gradient signals (hopefully) indicates “true” information
  - Optimization algorithms employ momentum, accumulating a trend in the gradient over many iterations, e.g.

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \frac{\eta_k}{n} \nabla Q_i(\mathbf{w}_k) + \alpha(\mathbf{w}_k - \mathbf{w}_{k-1})$$

- Adam (Adaptive moment estimation) adapts learning rate for each parameter, very popular choice
  - Better learning rate for sparser parameters





# Optimization with batches

- Overall goal: Good performance for general inputs, not just *overfitting* to training data → attempt to avoid rough model landscapes of many local minima
- Dropout
  - Randomly remove some activations at each training instance
  - Try to make sure that no single value controls the whole thing
- Batch normalization
  - Reshape activations to approximate a standard normal distribution
  - Based on the set of activations within the mini-batch



# Fully connected networks

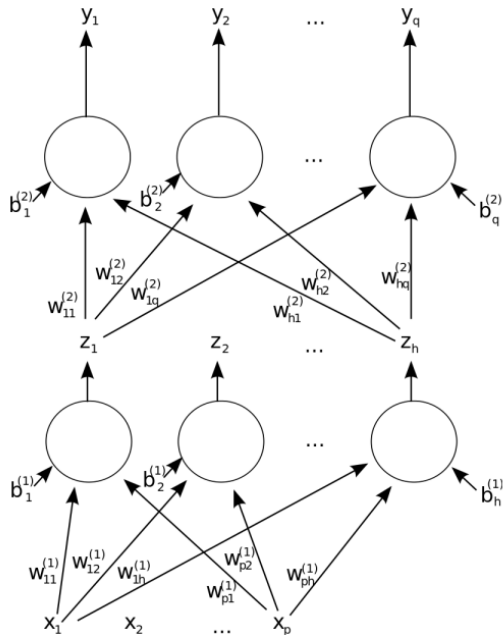
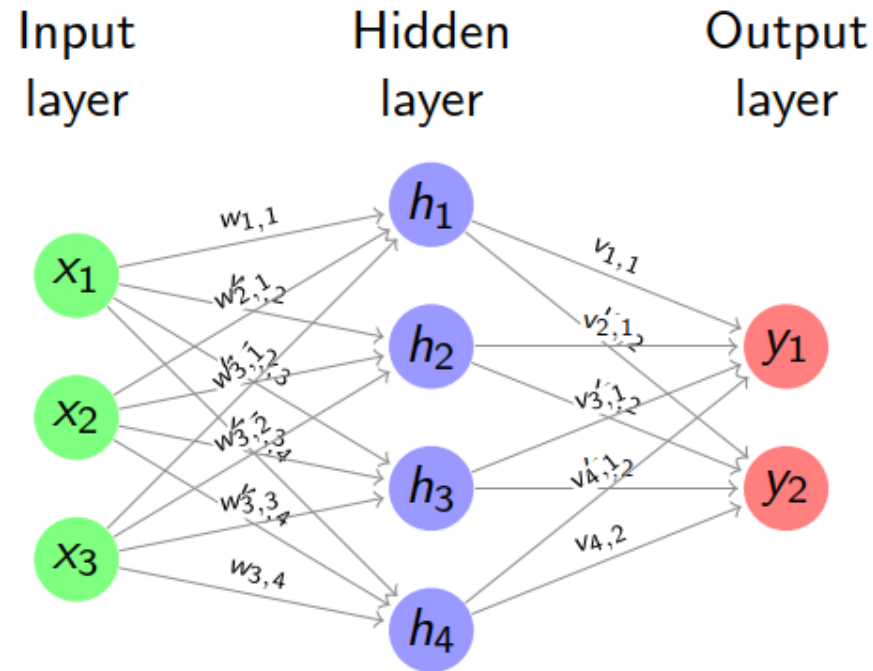


Image source:  
<https://commons.wikimedia.org/w/index.php?curid=9516009>

- Example is a fully connected network
  - Also called “dense”
  - Each node in each layer is connected to every node in the next layer by a weight
  - Leads to dense matrix-matrix Multiplication
- A fully connected network has a very high number of weights
  - Large model  $\rightarrow$  Slow to evaluate
- Hard to train
  - Lots of overfitting/local minima
  - Limited ability to generalize



# Example of mathematical operations in feedforward pass

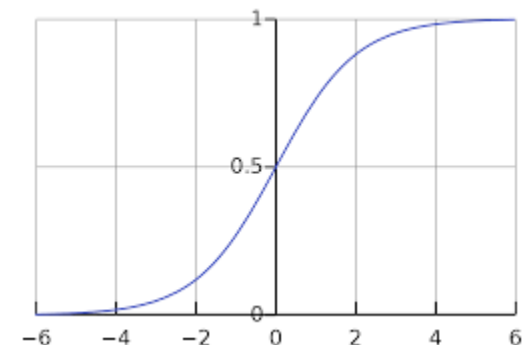


Multiply input by weights:

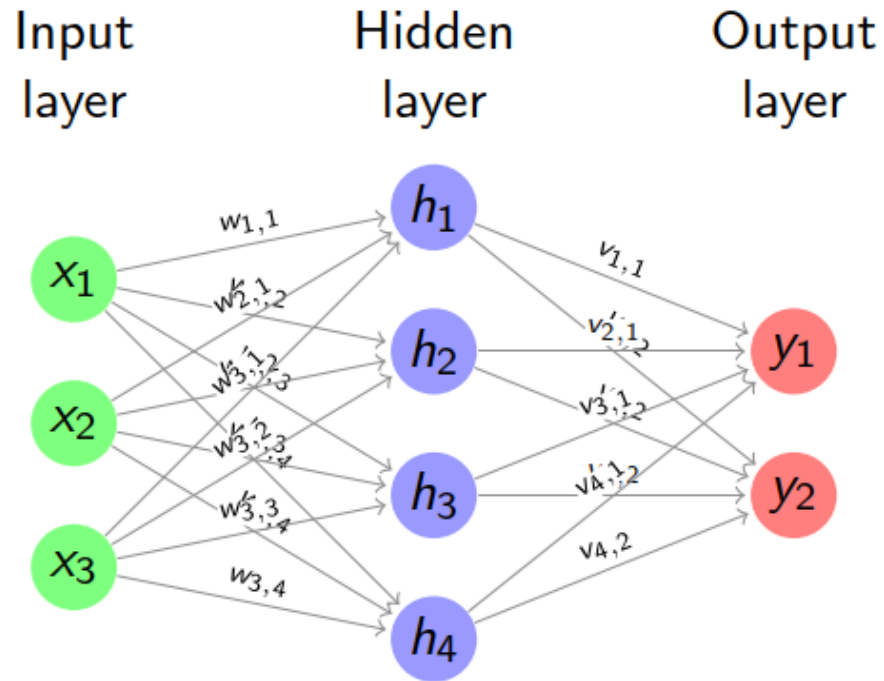
$$\begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \cdot \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} & w_{1,4} \\ w_{2,1} & w_{2,2} & w_{2,3} & w_{2,4} \\ w_{3,1} & w_{3,2} & w_{3,3} & w_{3,4} \end{bmatrix} = \begin{bmatrix} h'_1 & h'_2 & h'_3 & h'_4 \end{bmatrix}$$

Activation function in hidden layer e.g. sigmoid

$$\begin{bmatrix} h_1 & h_2 & h_3 & h_4 \end{bmatrix} = \frac{1}{1 + e^{\begin{bmatrix} -h'_1 & -h'_2 & -h'_3 & -h'_4 \end{bmatrix}}}$$



# Example of mathematical operations in feedforward pass



Next set of weights:

$$\begin{bmatrix} h_1 & h_2 & h_3 & h_4 \end{bmatrix} \cdot \begin{bmatrix} v_{1,1} & v_{1,2} \\ v_{2,1} & v_{2,2} \\ v_{3,1} & v_{3,2} \\ v_{4,1} & v_{4,2} \end{bmatrix} = \begin{bmatrix} y'_1 & y'_2 \end{bmatrix}$$

$$\text{Activation function } \begin{bmatrix} y_1 & y_2 \end{bmatrix} = \frac{1}{1 + e^{-\begin{bmatrix} -y'_1 & -y'_2 \end{bmatrix}}}$$



# Mathematical operations in backpropagation

Look at the operation for the last layer weights

$$v_{k+1} = v_k - \eta_k \frac{\partial Q}{\partial v} \quad \text{with cost func.} \quad Q(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$$

The  $\hat{y}$  with hat indicated training data,  $y$  is the forward model response. Using the chainrule we find

$$\frac{\partial Q}{\partial v} = \frac{\partial Q}{\partial y} \frac{\partial y}{\partial v} \quad \text{with the last derivative given by chainrule and def.} \quad y = f(h^T v)$$

$$\frac{\partial y}{\partial v} = \frac{\partial y}{\partial y'} \frac{\partial y'}{\partial v} = \frac{\partial f}{\partial y'} h^T$$

Previous layers are worked on successively, using the result of the previous derivatives, from right to left in the network



# Matrix-matrix multiplications in neural networks

- Feedforward model consists of matrix-vector product at each layer, combined with some activation function
- Similarly, the backpropagation for a single value  $y$  against measurement  $\hat{y}$  passes through the various layers one by one, involving matrix-vector products
- Typical models are evaluated (feedforward) or trained (backpropagation) for many sample sets
  - Backpropagation trains for many inputs/output pairs
  - matrix-vector  $\rightarrow$  matrix-matrix
- Concatenate many vectors into a matrix  $\rightarrow$  much improved data locality, matrix-vector products transformed in **matrix-matrix multiplications**
- Due to cacheability of matrix-matrix product ( $n^2$  data,  $n^3$  operations), much better performance possible



# Convolutional networks – ask the neighbor

- In a time series, or an image (or a combination, like a video), the absolute location is not crucial
  - Rather, compare to the adjacent samples
- A convolutional neural network is locally connected with shared weights
  - Apply a small (e.g. 3x3, 5x5) stencil to each pixel
  - Apply a shared set of weights to the pixels from the previous layer
  - If this is made deep, information can move far further than just one stencil width
  - Also reduction operations (averaging, max pooling) to reduce dimensionality
- → See Assignment 3



# Performance of convolutional neural networks

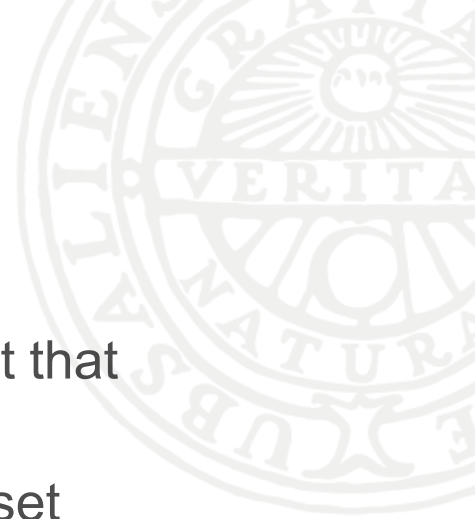
- Fully connected with  $N_1$  nodes connected to  $N_2$  nodes
  - Each of  $N_1$  activations is used  $N_2$  times
  - Lots of reuse of data
  - But reuse is global, how to parallelize efficiently?
- Convolution network
  - Each of  $N_1$  activations is used “stencil size” times
  - These are close together!
  - Idea: one GPU thread computes the activation from each new pixel
  - Adjacent GPU threads will access mostly the same data





# Generative adversarial network (GAN)

- Developed 2014
- Can get low mean-square error for reconstructing an image but still get a result that is obviously and clearly wrong by just looking at it
- Possible solution: Learn to generate new data with same statistics as training set
- Two neural networks contest with each other
  - Generator: the “main” network
    - Example: generate a face from a random seed vector
  - Discriminator: judge the result of the main network
    - Example answer: Is this a natural face?
- Discriminator is trained on true examples and generated ones
- Generator trained on output from discriminator
- Adversarial since they are trained for opposite goals
  - Discriminator adapts to identify typical generator errors
  - Generator’s goal is to fool discriminator
- Unsupervised learning



# Performance properties of GAN

- Two networks running at same time and locally generated data imply a high load of computations
  - Rather than (only) feeding in training data from an outside source, data is generated and used locally
  - More data locally on the GPU
- Adversarial networks have formed the basis for many of the most innovative developments during the last few years
  - Defining a good error metric used to be a hard challenge



# Performance of neural networks

- Central operation of most neural networks in both inference (feedforward) and training (backpropagation) is matrix-matrix multiplication (weight matrix: dense or sparse)
- How to make matrix-matrix multiplication fast?
- Long history in high-performance computing
  - Use of vectorization (SIMD) on CPUs, data re-use in registers most crucial (plus cache blocking for larger sizes)
  - Apply SIMT concept from GPUs
- Observation from practice
  - High precision in result  $w_{ij}$  not needed, result of optimization algorithm with number of layers/sparsity more important than many digits
  - Stochastic gradient insensitive to roundoff effects
  - Need only rough information (some digits)
- Idea: Reduce precision from  $\text{FP}_{64}$  (53 bits mantissa) to  $\text{FP}_{32}$  (24 bit mantissa),  $\text{FP}_{16}$  (11 bit mantissa),  $\text{BF}_{16}$  (binary float, exponent range of  $\text{FP}_{32}$  but only 8 bit mantissa), or even  $\text{INT}_{16}$  (-32767... 32767) or  $\text{INT}_8$  (-255... 255) in mat-mat
  - → Discussion of white paper



# Native instructions for matrix-matrix multiplications

- Given importance of matrix-matrix multiplication in neural network algorithms, want to make it fast
- Possibilities in classical reduced precision means less data transfer, allowing to transfer bigger “vectors” across a bus of fixed width
- Multiplication scales quadratically with the number of bits, halving number of bits means we can fit  $\sim 4\times$  the multiplications in same transistor budget
- Exploiting theoretical  $4\times$  improvements of low-precision multiplications bumps into limits of data transfer between registers and execution units  $\rightarrow$  traditional vector style execution sees rather  $2.x\times$  improvement
- Solution: Must perform matrix-matrix multiplication inside execution unit to avoid involving storage in registers
- (Tensorcore) Instruction for  $4 \times 4$ ,  $4 \times 8$ ,  $8 \times 8 \dots$  matrix multiplications
- Example  $n \times n$  matrix multiplication read accesses,  $n^2$  write access versus  $2n^3$  arithmetic operations; traditional vector-style involves  $n \times$  the transfer (from registers)



# Hardware for matrix-matrix multiplications

- NVIDIA GPUs: **tensor cores** → remember Ampere whitepaper
  - 2× higher throughput for double precision, 8 – 16× higher throughput for FP32/FP16, more for INT8
  - The wider the data, the smaller the size of matrices
- Next-generation CPUs scheduled to include matrix extensions
  - AMX on x86 (Advanced Matrix Extensions), to work on tiles, coming in 2023 (?)
  - Scalable matrix extensions for ARM
  - Apple chips also have “AMX2” unit, not directly accessible, only via frameworks like neural network or BLAS libraries
- Google’s **own tensor processing unit**, TPU (8 bit), specific interconnects
- They all target lower-precision computations with AI as main goal, slowly start to get used also for HPC
- Next step in **accelerator-based programming**
- NVIDIA tensor cores allow sparsity in matrices, given 2× speedup if at most 0.5× matrix is populated, to reflect sparse weights in convolutional networks



# TensorFlow: A framework for deep learning

- Library from Google
- Based on creating the computational graph through a syntax rather similar to using `numpy` in Python but not identical
- Working principle of TensorFlow: Given access to the full graph, can optimize memory transfers and evaluation order
- Lots of constructs specifically adapted to neural networks
  - Convolution operations, loss functions, optimizers
- But also a general evaluator of any computational graph
- Caveats:
  - TensorFlow may not always give optimal performance
  - Not every problem is reasonably expressed as computational graph (yet many are)



# TensorFlow: What is a tensor?

- In practice, a TensorFlow `tensor` is an object similar to a
- `numpy ndarray`
- A tensor has a dimensionality and a type
- Even in numpy, not every array-like object is backed by an actual chunk of memory
  - When slicing or transposing an array, that only creates a view inside another array
- In TensorFlow, many operations are just that, if we write

$$C = A * B,$$

`C` represents the operation of multiplying `A` and `B`, in their current state

- That operation might also be executed



# TensorFlow: Constants & variables

- To insert a value (scalar or an array) into TensorFlow, use
  - `tf.constant`, e.g.  
`tf.constant(someData, dtype=tf.float32)`
- To create a large tensor of identical values, most efficient
- approach is to use `tf.fill`
  - Accepting a scalar and a shape
- To create optimizable variables, use `tf.Variable`
  - These are end targets for gradient computations and optimization algorithms
  - The actual content of a model





# TensorFlow: Operations

- Operations are elementwise per default similarly to `numpy`
- `A * B`, `A + B`, `A / B` all act elementwise
- `A @ B` for matrix multiplication
  - syntactic sugar since Python 3.5, implies `tf.matmul`
- `tf.stack` combines several tensors of rank `R` into one of rank `R + 1`
- `tf.concat` concatenates several tensors of rank `R` into a new tensor of rank `R` with larger dimensions
- `tf.where` takes a boolean tensor choosing elements from the
- arguments, e.g. maximum by `tf.where(A>B, A, B)`
  - `tf.math.maximum(A, B)` is more convenient and probably more efficient
  - `tf.math.reduce_max(A)` instead computes the maximum within a tensor



# TensorFlow: General arguments

- Almost all TensorFlow functions accept a name argument
- TensorFlow itself does not “see” user-given variable names, so error messages can sometimes refer to layer names
- Many functions can work on the full tensor, or just along some axis
  - Operation mode can be changed using the axis argument
  - `tf.reduce_sum` has a default axis of `None`, summing over all axes
  - `tf.concat` and `tf.stack` have defaults of `0`, indicating that the common/new axis should be the first one
  - Negative indices are allowed, just like in ordinary Python, to access indices starting from the end



# TensorFlow: Broadcasting

- TensorFlow supports broadcasting similarly to `numpy`
  - If a tensor is size 1 in some dimension, that can implicitly be interpreted to match any other size for many operations
- Risk for hard-to-understand errors due to this implicit conversion
  - Multiplying a shape  $N \times 1$  vector ( $N$  rows, 1 column) with a  $1 \times N$  vector (1 row,  $N$  columns) creates a  $N \times N$  matrix
- `tf.broadcast` to, `tf.reshape` and `tf.expand_dim` can be useful when you need a bit more control
- Broadcasting is far more efficient than actually creating the corresponding tensor with repeated elements
  - In abstract setting, can avoid multiplication with repeated entries and mathematically “transform” results
- The same holds for `tf.tile`, for repeating a tensor multiple times



# Eager execution

- Traditional workflow with TensorFlow: first create a graph and then ask TensorFlow to run to get the value of some specific tensor
  - Only viable usage mode in TensorFlow 1
  - Create the graph, then run it
- TensorFlow 2 supports eager execution, where the operations you do are also evaluated
  - Does not build graphs
  - TensorFlow runs in eager mode by default, check by `tf.executing_eagerly()`
- Much easier to debug this way. . .
  - This approach removes some optimization opportunities



`@tf.function`

- TensorFlow allows to decorate a function by `@tf.function`
- This will make TensorFlow analyze the full function and try to express it as TensorFlow graphs
- This gives room for optimization
- Sometimes even `for` loops and other “expensive” things in Python can be pushed into a compact graph that is executed all on the GPU
- Enclose well-contained logic in functions (good practice anyway), and tag them as `@tf.function`, unless need to debug them



# TensorFlow: Pitfalls

- A tensor can stay on the GPU
  - Depends on the calling context and other dependencies
- TensorFlow can be very helpful in converting to and from `numpy` arrays
  - This will transfer the information between CPU and GPU
  - All gradient information will be lost when converting from a tensor to `numpy`, do something, and then transforming back
    - From TensorFlow's point of view, those are two unrelated constant tensors
    - Abstract tensor notation lost, transformed into numbers
- Even just doing an `if` on some value of a tensor outside of a `@tf.function` forces the full evaluation of the tensor and the transfer of that data from GPU to CPU
- It is so easy to do things with TensorFlow that the cost of certain operations gets hidden and thus non-obvious



# Auto-differentiation with TensorFlow

- Backpropagation would be terrible to use if implemented manually
- TensorFlow provides automatic computation of gradients
  - Automatic for variables, but can compute the gradient also for a constant
- Consequence: Almost any TensorFlow operation is differentiable
  - But the gradient of e.g. a `max` operation is only related to the maximum value
  - Even if a variable is lower by only a factor  $1 - 10^{-5}$ , it gets zero gradients
- This and other discrete operations typically avoided
- Aim rather for mathematically smooth operations, like computing a norm, or taking the sum of the logged exponentials



# TensorFlow: Code example



```
x = tf.constant(3.0)
with tf.GradientTape(persistent=True) as g:
    g.watch(x)
    y = x * x
    z = y * y
dz_dx = g.gradient(z, x) # 108.0 (4*x^3 at x = 3)
dy_dx = g.gradient(y, x) # 6.0
del g # Drop the reference to the tape
```





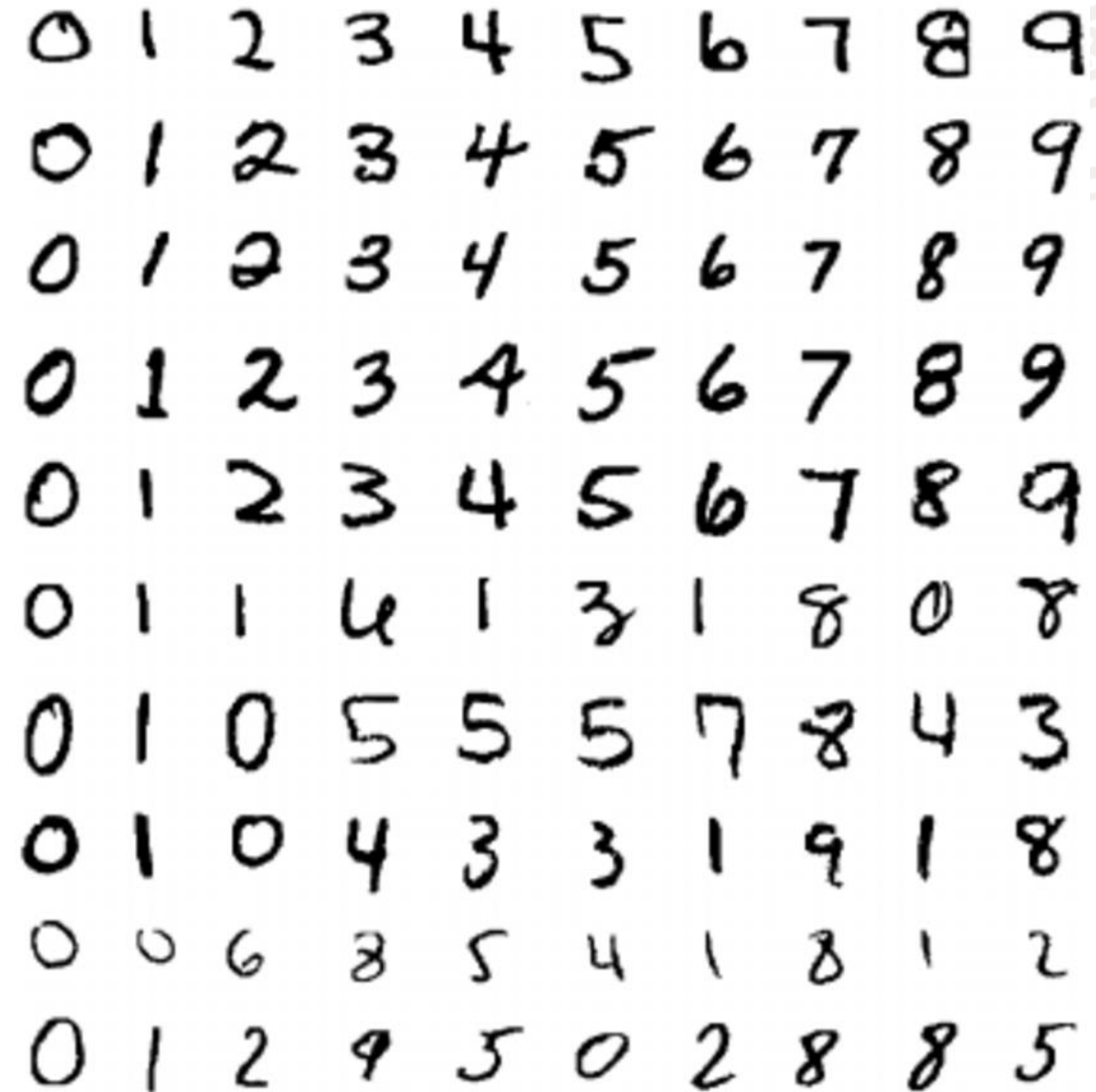
# Keras: A higher-level abstraction for neural networks

- Keras is a higher level abstraction for building neural networks
- Goal: Fast experimentation with deep neural networks
- Has Python interface and interface to TensorFlow
  - In theory, Keras can be used with several backends (not only TensorFlow), but it can also just be used as a convenience layer
- Keras provides numerous implementations of neural-network building blocks:
  - Objectives, layers, activation functions, optimizers
  - Support for convolutional and recurrent neural networks
- Sometimes confusing to find Keras and TensorFlow features to do the same thing



# Lab exercise with Keras+Tensorflow

- MNIST data set of 28x28 pixel images for classification of handwritten digits
- Neural network of 128x128x10 layers
- Classification



0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9
0	1	1	4	1	3	1	8	0	8
0	1	0	5	5	5	7	8	4	3
0	1	0	4	3	3	1	9	1	8
0	0	6	8	5	4	1	8	1	2
0	1	2	9	5	0	2	8	8	5

# Lab assignment 3 with Tensorflow

- Conway's game of life using GPU convolutions.
- Cellular automaton
- Rules:
  1. Any live cell with two or three live neighbors survives.
  2. Any dead cell with three live neighbors becomes a live cell.
  3. All other live cells die in the next generation. Similarly, all other dead cells stay dead.

Can be efficiently implemented using convolutions → fast on GPU as independent for pixel

