UPPSALA
UNIVERSITET

# GPUs

*Carl Nettelblad*

*2020-06-01*

UPPSALA UNIVERSITET

# Welcome to the course!

- A course on how to scale machine learning models
  - And other computations.
- How do we scale from CPU to GPU?
  - Why does it work?
  - When won't it work?
  - How can we roll our own algorithms?
- How do we scale from *development* on single nodes to *production workflows* in a cloud?

- A project where you can try these techniques out yourself.

UPPSALA UNIVERSITET

# Who am I?

- Carl Nettelblad
  - PhD in Scientific Computing 2012
    - Main research in statistical methods for life science
    - Much implemented on CPUs, but interests include deep learning, and other processing, on GPUs
  - Supervised my first student project using CUDA in 2009
  - Currently associate professor at Department of IT, UU, as well as technical coordinator at UPPMAX
  - A past as a competitor in programming competitions, including the International Olympiad in Informatics, the ICPC world finals, and TopCoder Open
  - I love the intersection of theory, algorithms, and applications. I strongly believe the best science is made when you take a small step outside of run-of-the-mill approaches in all three.

# Please!

- After the first break, I'll ask you all to just say some quick words about yourself.

- Who are you? What department/university? Are you taking this course as a way to broaden your views, or do you see specific connections to your research?

UPPSALA UNIVERSITET

# Interaction!

- Please interrupt!
- It is harder to “read the room” and see what needs clarification or not, when there is no room.
- You all have diverse experience and knowledge that is worth sharing.

UPPSALA UNIVERSITET

# What's a CPU?

- A von Neumann machine
- Memory
  - Instructions and data
- CPU
  - Read instruction from current instruction counter, increment counter
  - Instruction might request data from memory
  - Instruction might modify instruction counter
  - Repeat until turned off

UPPSALA UNIVERSITET

# What's a CPU?

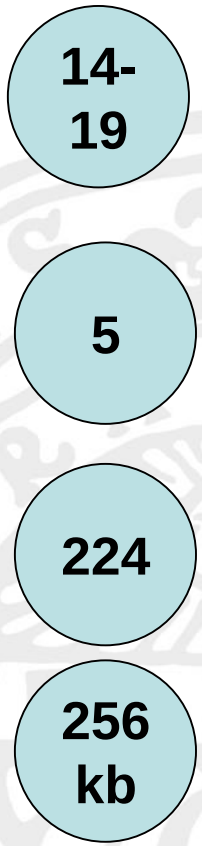
- This is a fairytale.
- Very early CPUs worked something like this.
- One instruction at a time, instruction *i* has to complete before executing instruction *i* + 1.

UPPSALA UNIVERSITET

# Why is this not true?

**Intel Skylake**

- The CPU is:
  - Pipelined, several instructions are “in flight” at once
    - Read and decode instruction *i + 2*
    - Compute the result of instruction *i + 1*
    - Store the result of instruction *i*
  - Superscalar, dispatch several instruction per clock cycle
    - Several pipeline steps are “wide”, allowing multiple instructions to be handled at once
  - Out-of-order
    - If instruction i can not complete due to waiting for data, or the result of another instruction, start instruction i + 1 instead
  - Cached
    - Main memory is slow, store copies locally
  - All of these had happened by the year 2000!



UPPSALA UNIVERSITET

# But there's more…

- The vendors added more and more logic to keep the illusion of the von Neumann machine.
- All these steps still maintain the illusion of a single stream of instructions (a single thread) executing at once.
- This also includes tricks like *branch predictors* and *prefetchers*.
  - You need to know the result of an `if` to know what the next instruction is
  - You want to guess what indexes will be accessed in order to (ideally) bring those into the cache before they are accessed!
- Mispredictions and cache misses are expensive.
  - Seemingly innocent computer science concepts like virtual methods and linked lists can get expensive when realized on a modern architecture

UPPSALA UNIVERSITET

# What happened after 2000?

- Frequencies topped out at 5 GHz
- It got exceedingly hard to keep a single thread running fast

- Add more cores
  - Don’t just add execution units within the core, but fully separate CPUs within the same socket
  - 4-8 common in desktops and high-end laptops now
  - Add simultaneous multi-threading, 2-4 threads on the *same* core
- Add more cache
  - Shared between cores, L3
- Add wider *vector* instructions
  - Process more 32-bit numbers at the same time
  - 4… 8… 16… 32
  - But even HPC codes have a hard time to use these

UPPSALA UNIVERSITET

# Single core CPU die shot


From wikichip

UPPSALA UNIVERSITET

# This is what makes CPU great!

- They are (comparatively) “easy to program” for the very reason that most of the silicon is spent on hiding problems and speeding up the simple von Neumann abstraction

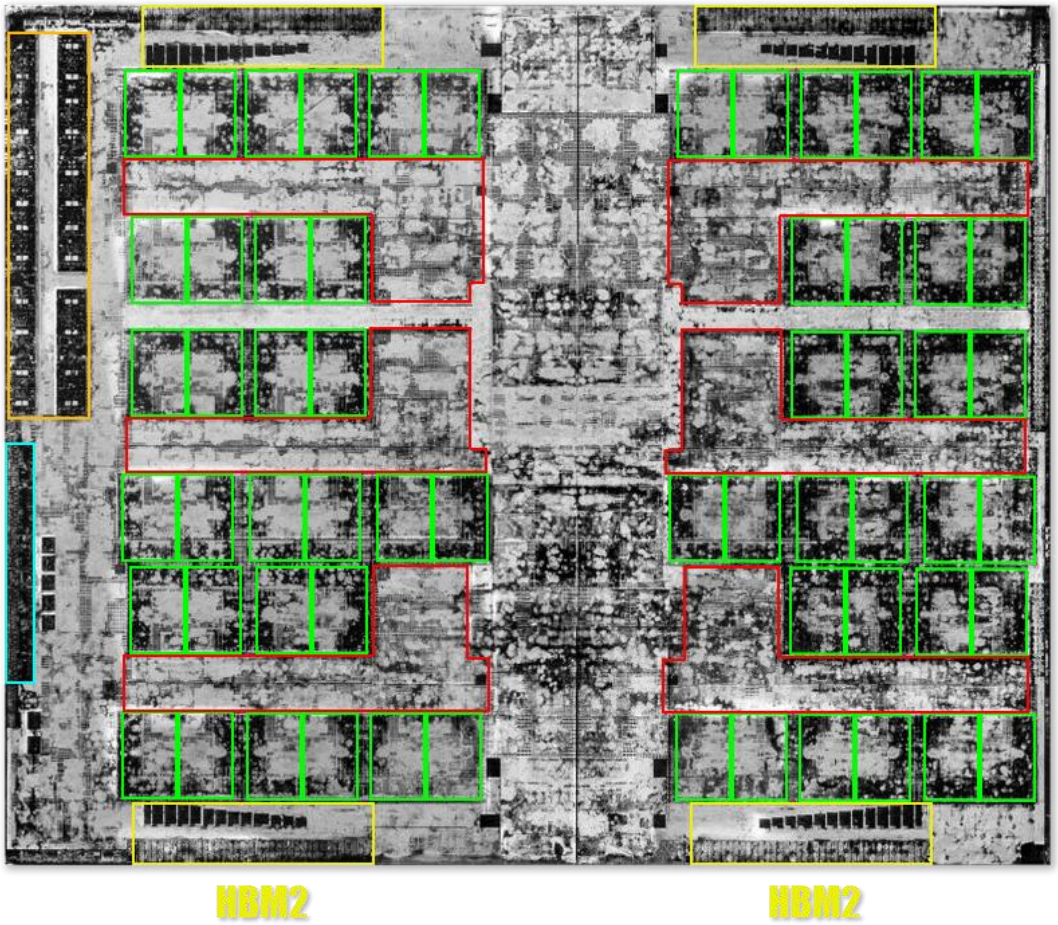
UPPSALA UNIVERSITET

# What's a GPU?

- Render pixels
  - In the year 2000, 1024x768 at 60 FPS
  - Done by drawing triangles
  - Compute the position of each triangle, compute the color of each pixel
- Extremely parallel
- Made more flexible over time, but the main principle stands
  - There is *not* a single main thread or a few of them
  - There are thousands of threads

UPPSALA UNIVERSITET

# GPU die shot




60x
Multiprocessors

6x
Thread Dispatch
Setup Pipes (?)

UPPSALA UNIVERSITET

# The GPU is limited

- Not out of order
- No branch prediction
- Very limited automatic prefetching
- Some caches
- Lots of simultaneous multithreading
  - When stalling (waiting for operation to complete), switch to another thread
- So, if you have 1000 “cores”, you can easily want to have 8000 threads
- **Does your algorithm have 8000 threads?**

UPPSALA UNIVERSITET

# Memory

- Normal system memory is slow (maybe 30 GB/s/CPU)
- A GPU can easily have 300 GB/s
- Lower latency as well
- Again, the CPU cores create this fairytale world
  - Out of order, branch prediction, prefetching, lots of cache space *per thread*
  - And much of caches local per core
    - You want to find independent tasks with independent data

UPPSALA UNIVERSITET

# Comparison

- A Rackham node
  - 20 cores
    - At most 40 simultaneous threads
  - 128 GB system memory
  - ~60 GB/s memory bandwidth
  - 5120 kB core-specific cache (only counting L2)
  - 250+ W just for CPUs

UPPSALA UNIVERSITET

# Comparison

- Tesla T4
  - A lightweight current-generation GPU
  - A gaming card redressed for servers
  - 2560 cores
    - At most 40,960 simultaneous threads
  - 16 GB video memory
  - ~300 GB/s memory bandwidth
  - 3840 kB SM-specific memory/cache in total
    - Shared by 64 cores
  - 70 W per GPU

UPPSALA UNIVERSITET

# Resources per thread

| Resource/thread | Rackham CPU | Snowy GPU |
|---|---|---|
| Cores | 0.5 | 0.0625 |
| Memory | 3250 MB | 0.4 MB |
| Memory bandwidth | 1500 MB/s | 7.5 MB/s |
| Cache | 131 072 bytes | 96 bytes |
| Scalar SP FLOPS (base) | 2200 GFLOPS | 73 GFLOPS |
| Scalar SP FLOPS (boost) | 3400 GFLOPS | 200 GFLOPS |
| Vector SP FLOPS (base) | 14 400 GFLOPS | 73 GFLOPS |
| Vector SP FLOPS (boost) | 24 800 GFLOPS | 200 GFLOPS |

UPPSALA UNIVERSITET

# Math

- Originally, GPUs focused on rendering RGB pixels, 3*8-bit colors
- Later generalized to floating point
  - FP good for all 3D transforms
- These days, they can do proper integer math just as fast
- Text is just integers
- Memory addresses are also integers

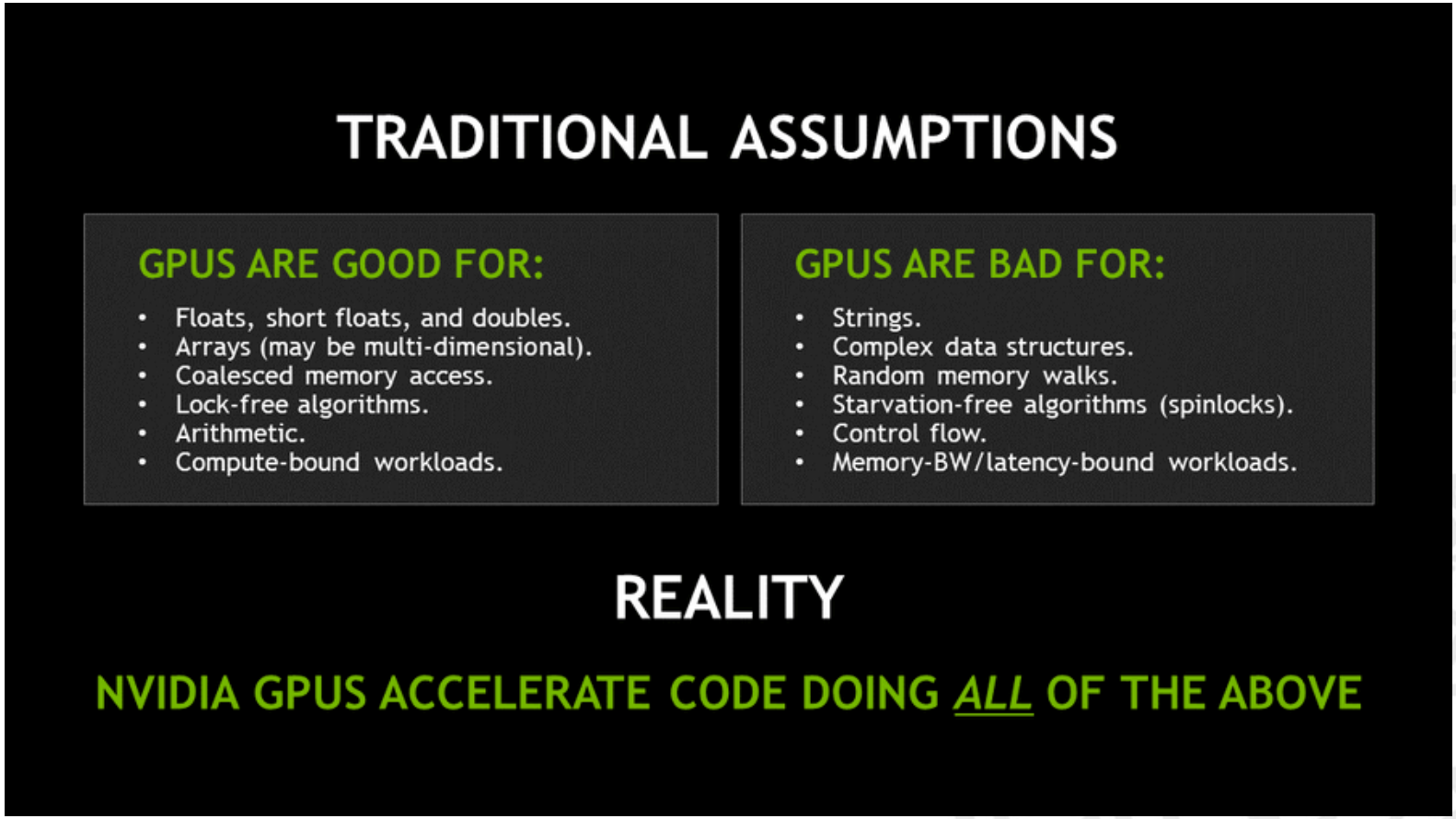
UPPSALA UNIVERSITET

# Summary

- An extremely parallel computational unit
  - But fully programmable
- For CPU
  - Keep threads separate
  - Thread count ~ core count
  - Communication is expensive
  - Vector instructions for high performance
- For GPU
  - Each thread is weak
  - Thread count >> core count
  - Keep multiple threads working in concert
  - Groups of threads mainly replace vector instructions

UPPSALA UNIVERSITET

# Nvidia's take on this



UPPSALA UNIVERSITET

# My take on Nvidia

- The left-hand box is *easier* to make fast
- But those algorithms also shine on CPUs
  - GPUs excel at dense matrix multiplications and multi-dimensional Fourier transforms
  - These algorithms have repeated passes over the same data
  - But CPU vector instructions also do reasonably well
- The single most important thing is extreme parallelism
  - *Relative to other resources*
  - The relative benefit to CPU can be the greatest when the algorithm is hard to coax into vector instructions (manually or using compiler)
    - Even if resource utilization of GPU gets abysmal

# Please!

- After the first break, I'll ask you all to just say some quick words about yourself.

- Who are you? What department/university? Are you taking this course as a way to broaden your views, or do you see specific connections to your research?

UPPSALA UNIVERSITET

# Deep learning

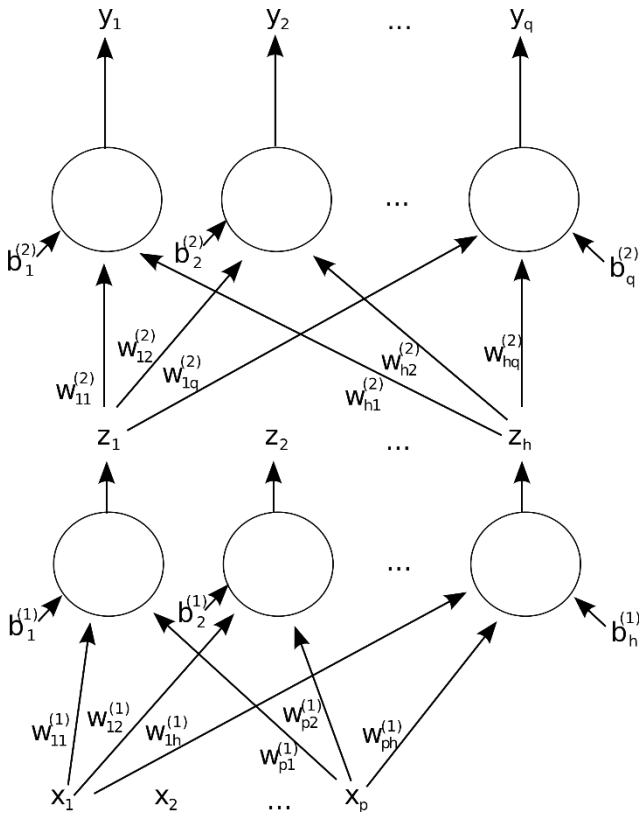
- We focus more on the computational machinery in this course then actual models
- Still relevant to have some familiarity
- I know several of you actively work with deep models
  - I welcome discussion and clarifications

UPPSALA UNIVERSITET

# Deep learning

- Artificial neural networks have existed since the 1960s
  - “Multi-layer perceptrons” were popular in the 1980s
- Deep learning refers to the fact that the number of layers has increased significantly

# Some layers




By Mcstrother - Own work, CC BY 3.0, https://commons.wikimedia.org/w/index.php?curid=9516009

- A simple two-layer network, inputs x, outputs y
- Inputs are multiplied by weights
  - Summed
- Bias added
- Activation function applied
- Result from activation functions forms output, fed to the next layer

UPPSALA UNIVERSITET

# Graph

- The network is a directed (acyclic) graph
  - This is called a feed-forward network
  - General, biological, neural circuits do not need to behave like this
    - But in fact, some important biological structures *do*
- The edges indicate the flow of data

UPPSALA UNIVERSITET

# Backpropagation

- Define your desired outputs (y in our example)
- Define how to compare outputs against the desired outputs (objective function/loss function)
  - Mean square error is a simple example

- How can we minimize the loss?
  - Compute the gradient
  - If you know the activation and the gradient for the output step, you 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

UPPSALA UNIVERSITET

# Mini-batches

- The training dataset is frequently very large
- To do backpropagation, you need to store the activation value for every single node in the network, for every input
  - This consumes space and bandwidth
- Especially on GPUs, it is therefore crucial to instead train in batches
  - More samples in a batch might allow greater compute parallelism
  - More samples in a batch can exhaust memory and reduce cache performance

UPPSALA UNIVERSITET

# Optimization

- Just tracking the gradient seems simple
  - But it's extremely slow
    - Small steps (low *learning rate*) not to overshoot
  - Function not convex
    - Moving towards the top of a hill won't bring you to Mount Everest

UPPSALA UNIVERSITET

# Optimization

- Different batch configurations and optimization algorithms try to avoid this
  - If you use different random subsets for each batch, hopefully the stable gradient signals indicate “true” information
  - Optimization algorithms employ momentum, accumulating a trend in the gradient over many iterations
  - Adam (Adaptive momentum) a very popular choice

UPPSALA UNIVERSITET

# Optimization

- Attempts to avoid rough model landscapes of many local minima and *overfitting* to only training set, not general behavior
  - Dropout
    - Randomly removing some activations at each training instance
    - Try to make sure that no single value controls the whole thing
  - Batch normalization
    - Reshaping activations to approximate a standard normal distribution
    - Based on the set of activations within the mini-batch

UPPSALA UNIVERSITET

# Convolution

- Our example was a fully connected network
  - Also called “dense”
  - Each node in each layer is connected to every node in the next layer by a weight
- A fully connected network will have a very high number of weights
  - Large model
    - I.e. slow
  - Hard to train
    - Lots of overfitting/local minima
    - Limited ability to generalize

UPPSALA UNIVERSITET

# Ask your 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

UPPSALA UNIVERSITET

# What does this mean for performance?

- Fully connected
  - Each of $N_1$ activations is used $N_2$ times
  - Lots of reuse of data!
  - But reuse is global
- Convolutional
  - Each of $N_1$ activations is used *stencil size* times
  - These are close together!
  - You can imagine one GPU thread computing the activation from each new pixel
  - Adjacent GPU threads will access mostly the same data

UPPSALA UNIVERSITET

# Adversarial networks

- You can get a low mean-square error for reconstructing an image, and still get a result that is obviously and clearly wrong if you just look at it
- One option is to train multiple networks
  - Generator – the “main” network
    - Generate a face from a random seed vector
  - Discriminator – judge the result of the main network
    - Answer: Is this a natural face?
- Discriminator is trained on true examples and generated ones
- Generator trained on signal from discriminator
- Adversarial since they are trained for opposite goals
  - Discriminator adapts to identify typical generator errors
  - Generator adapts to fool discriminator

UPPSALA UNIVERSITET

# What does this mean for performance?

- Lots of computations going on
- More data locally on the GPU
- Rather than (only) feeding in training data from an outside source, data is generated and used locally

- 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

UPPSALA UNIVERSITET

# Pre-lab 1

- What results did you get?

- Can you relate those to what we have talked about? (What kind of model did we use?)

UPPSALA UNIVERSITET

# TensorFlow

- Library from Google
- Based on creating the computational graph through a syntax rather similar to using `numpy` in Python
  - But not identical
- If you have access to the full graph, you 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

UPPSALA UNIVERSITET

# What is a `Tensor`?

- In practice, a TensorFlow tensor is an object similar to a `numpy ndarray`.
- It has a dimensionality and a type.
- Even in `numpy`, not every array-like object is backed by an actual chunk of memory.
  - If you slice or transpose an array, that only creates a view inside another array.
- In TensorFlow, many operations are just that, if you write `C = A * B`, C represents the operation of multiplying `A` and `B`, in their current state.
  - That operation might also be executed.

UPPSALA UNIVERSITET

# Constants, variables

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

UPPSALA UNIVERSITET

# Operations

- Just like in `numpy`, operations are elementwise per default.
- `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
- `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

UPPSALA UNIVERSITET

# General arguments

- Almost all TensorFlow functions accept a `name` argument. TensorFlow itself doesn't "see" your variable names, so error messages can sometimes refer to layer names.
- Many functions can work on the full tensor, or just along some axis. The 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.

UPPSALA UNIVERSITET

# Eager execution

- You might read a lot of stuff on how one first creates a graph and then asks TensorFlow to run to get the value of some specific tensor
  - This was the 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
  - This is *much* easier to debug… but it removes some optimization opportunities

UPPSALA UNIVERSITET

# `@tf.function`

- As stated, TensorFlow will run in eager mode by default
- You can 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 you want to debug them

UPPSALA UNIVERSITET

# Pitfalls

- A tensor can stay on the GPU
- TensorFlow can be very helpful in converting to and from `numpy` arrays
  - If you do that, the information will be transferred CPU<->GPU
  - All gradient information will be lost if you convert from a tensor to numpy, do something, and then transform back
    - From TensorFlow's point of view, those are two unrelated constant tensors
- 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's so easy to do some things that it's not clear that some things are expensive

UPPSALA UNIVERSITET

# Broadcasting

- Like `numpy`, TensorFlow supports broadcasting
  - If a tensor is size 1 in some dimension, that can implicitly be interpreted to match any other size for many operations
- Also creates risk for hard-to-understand errors
- Multiplying a shape (1,N) vector with a (N,1) vector creates a (N,N) matrix
- `tf.broadcast_to`, `tf.reshape` and `tf.expand_dims` can be useful when you need a bit more control
- Broadcasting is far more efficient than actually creating the corresponding tensor with repeated elements
- The same holds for `tf.tile`, for repeating a tensor multiple times

UPPSALA UNIVERSITET

# Keras

- In the prelab, you saw an example of Keras.
- Keras is a higher level abstraction for building neural networks.
  - In theory, Keras can be used with several backends (not only TensorFlow), but it can also just be used as a convenience layer
- Sometimes confusing to find Keras and TensorFlow features to do the same thing.

UPPSALA UNIVERSITET

# Auto-differentiation

- Backpropagation would be terrible to use if you would implement it manually
- TensorFlow provides automatic computation of gradients
  - Automatic for variables, but you can compute the gradient for a constant as well

UPPSALA UNIVERSITET

# Consequences of differentiation

- (Almost) *any* TensorFlow operation is differentiable.
  - But the gradient of e.g. a max operation is only related to the maximum value. Even if the lower value is only a factor 1-1e-5 smaller, it's ignored.
  - This and other discrete operations is something you want to avoid. Rather, one tries to have "soft" operations, like computing a norm, or taking the sum of the logged exponentials.

UPPSALA UNIVERSITET

# 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
```

UPPSALA UNIVERSITET

# Other frameworks

- PyTorch
  - An early main difference was that PyTorch was more like the eager mode in TensorFlow 2
  - Somewhat easier to use for distributed training over multiple GPUs
  - Almost a matter of taste these days
  - Many models can be found for TensorFlow as well as PyTorch
  - Sometimes higher performance in benchmarks
- MXNet
  - More advanced and varied GPU support
  - Frequently higher performance in benchmarks
  - The basis for Minpy, that we will try tomorrow
  - Not as widespread in research or industrial usage, harder to find good online resources

UPPSALA UNIVERSITET

# Lab 1

- Game of Life, created by John Horton Conway in 1970
  - Conway passed away in Covid-19 earlier this year
- Discrete Cartesian grid of cells
  - Each grid cell has 8 neighbors
  - Each cell can be alive or dead

UPPSALA UNIVERSITET

# Standard rules

- Any live cell with two or three neighbors survive
  - All other live cells die
- Any dead cell with three live neighbors gets alive
  - All other dead cells stay dead

- All rules evaluated based on number of live/dead neighbors in previous timestep

UPPSALA UNIVERSITET

# Naïve algorithm

- Compute number of live neighbors to each cell on a fixed grid
  - Possibly with extra term for current cell alive/dead
- Apply rule based on live/dead status and number of live neighbors
- This can be represented as a convolution
- Repeat for N generations

UPPSALA UNIVERSITET

# Hashlife

- Just for reference, the most advanced algorithm uses the fact that Game of Life is fully deterministic
  - If you find that a certain patch of the universe is identical to some patch you've seen before, you can predict the evolution of that patch
  - With some caveats of course
- We will not look into this

UPPSALA UNIVERSITET

# Example

- https://en.wikipedia.org/wiki/Breeder_(cellular_automaton)#/media/File:Conways_game_of_life_breeder_animation.gif
- https://upload.wikimedia.org/wikipedia/commons/e/e6/Conways_game_of_life_breeder_animation.gif

UPPSALA UNIVERSITET

# Q&A

- See you at 1.15 PM
  - You don’t have to stay in the Zoom room all the time
  - I’ll try to make sure that I give notice on Slack if we are about to discuss something of interest to a lot of people
- Feel free to start with the lab before that
- Note that the reservation is only during 1.00-5.00, so if you want to work on it outside that time, do not use the `--reservation` flag