Lesson 9 - Loss functions, optimisers & the training loop

Thomas Keil & Liad Magen

6th November 2019

Agenda

- Mathematical basics - short recap
- Data pre-processing and weights initialization
- Neural Net Forward Path
- Loss functions
- Back propagation
- Decorators
- Coding Best-practices

Mathematics for deep learning - recap

- Motivation:
  - Core concepts in machine learning
  - Optimising loss functions → calculus
  - Vectorisation techniques → computation efficiency
  - Integer overflow/underflow problem solutions (matrix decompositions)

- Mathematical entities:
  - Scalars
  - Vectors
  - Matrices
  - Tensors

Mathematics for deep learning - recap

- Important mathematical operations
  - Matrix addition / subtraction
  - Matrix multiplication
  - Matrix inverse
  - Matrix transpose
  - Hadamard product $(A \circ B)_{ij} = (A \odot B)_{ij} = (A)_{ij}(B)_{ij}.$

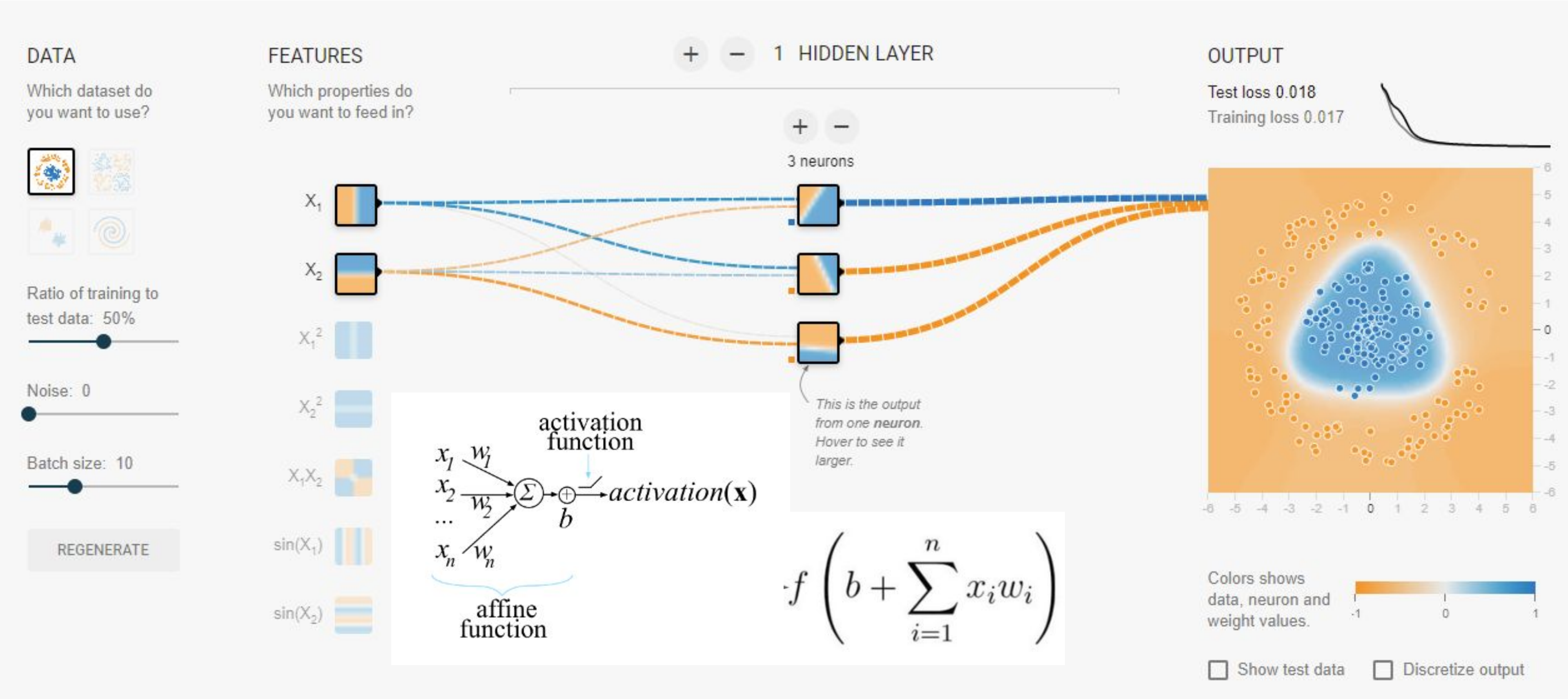
- Mathematical properties
  - Associative
  - Commutative
  - Distributive

Mathematics for deep learning - recap

- Multivariate calculus
  - Finding optimized solutions
  - Gradients calculation
  - Jacobian matrix
  - Chain rule

Rule	$f(x)$	Scalar derivative notation with respect to x	Example
Constant	c	0	$\frac{d}{dx}99 = 0$
Multiplication by constant	cf	$c\frac{df}{dx}$	$\frac{d}{dx}3x = 3$
Power Rule	x^n	nx^{n-1}	$\frac{d}{dx}x^3 = 3x^2$
Sum Rule	$f + g$	$\frac{df}{dx} + \frac{dg}{dx}$	$\frac{d}{dx}(x^2 + 3x) = 2x + 3$
Difference Rule	$f - g$	$\frac{df}{dx} - \frac{dg}{dx}$	$\frac{d}{dx}(x^2 - 3x) = 2x - 3$
Product Rule	fg	$f\frac{dg}{dx} + \frac{df}{dx}g$	$\frac{d}{dx}x^2x = x^2 + x2x = 3x^2$
Chain Rule	$f(g(x))$	$\frac{df(u)}{du}\frac{du}{dx}$, let $u = g(x)$	$\frac{d}{dx}ln(x^2) = \frac{1}{x^2}2x = \frac{2}{x}$

Recap - Network Structure



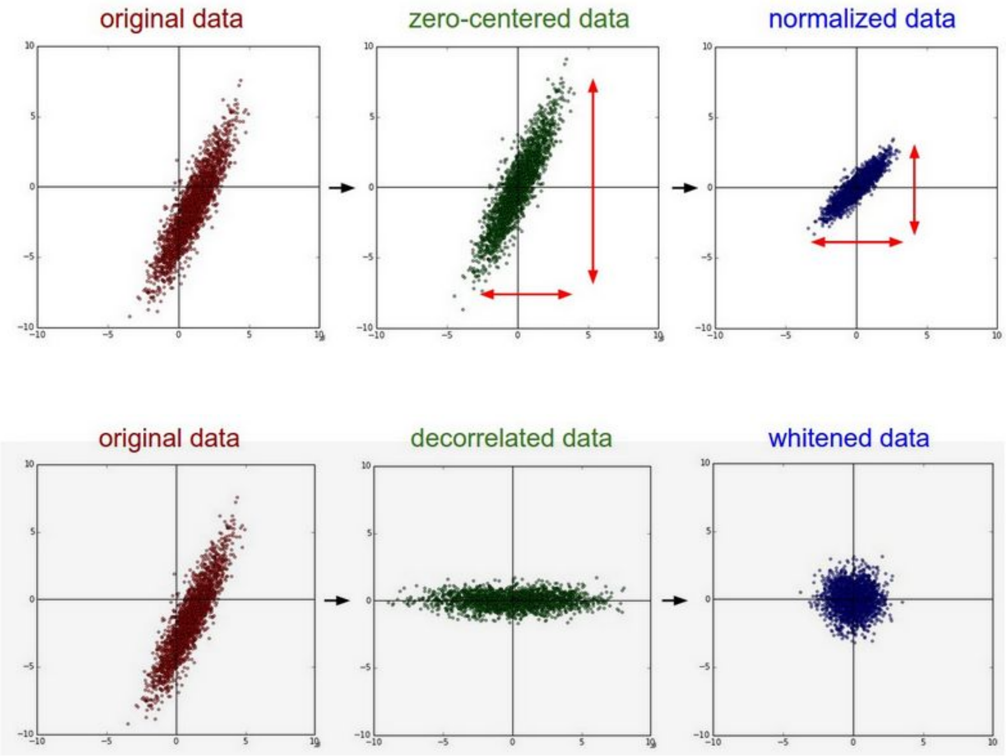
Data pre-processing

- Rationale / Motivation
    - Normalising dataset to the same statistical scale (equal importance)
    - Dimension reduction of only relevant data (computer efficiency)

- Methods
    - Zero-centering by first moment adjustment (e.g. mean)
    - Normalising statistical deviations by second moment adjustment (e.g. standard deviation)
    - Dimension reduction by using matrix factorisation and relevance (e.g. SVD, PCA)

- Word of caution
    - Never pre-process on test data
    - Ask yourself if normalising is necessary - equal importance of all features?
    - Certain dimension reduction techniques require statistical properties (e.g. linearity, gaussian)

Data pre-processing



https://cs231n.github.io/neural-networks-2/

Network weights initialization

- Rationale / Motivation
  - Prevent layer activation outputs from exploding/vanishing
  - Integer overflow/underflow problem
  - Effect on gradient size on back propagation

- Variance properties
  - Tower law of the expectation and variance
  - Additive summation over the layers

- Solutions
  - Random Initialization
  - 0-initialization
  - Xavier Initialization
  - **Kaiming He Initialization**

$$\mathrm{Var}(s) = \mathrm{Var}(\sum_i^n w_i x_i)$$

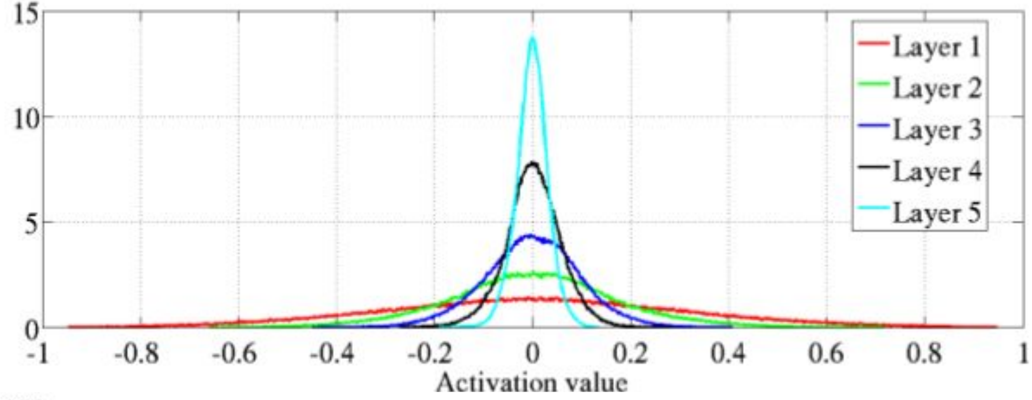
$$= \sum_i^n \mathrm{Var}(w_i x_i)$$

$$= \sum_i^n [E(w_i)]^2 \mathrm{Var}(x_i) + E[(x_i)]^2 \mathrm{Var}(w_i) + \mathrm{Var}(x_i)\mathrm{Var}(w_i)$$

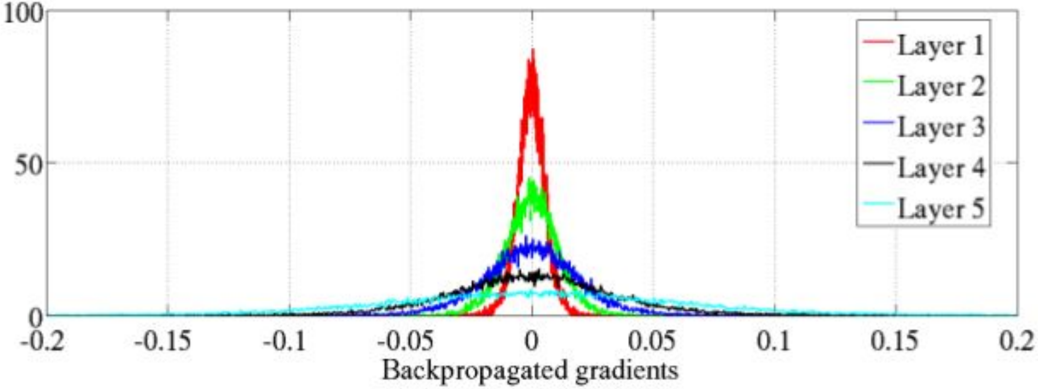
$$= \sum_i^n \mathrm{Var}(x_i)\mathrm{Var}(w_i)$$

$$= (n\mathrm{Var}(w))\,\mathrm{Var}(x)$$

Weights initialisation visualised



Average activation values with hyperbolic tangent activation (tanh) after standard initialisation



Average back-propagated gradients with hyperbolic tangent activation (tanh) after standard initialisation

Glorot, Xavier & Bengio, Y.. (2010). Understanding the difficulty of training deep feedforward neural networks. Journal of Machine Learning Research - Proceedings Track. 9. 249-256.

Training loop - Forward Path

For every neuron, in every layer:

$$f\left(b + \sum_{i=1}^{n} x_i w_i\right)$$

Activation functions:

- Sigmoid function
- Tanh function
- **ReLU function**

```
lr = 0.5  # learning rate
epochs = 2  # how many epochs to train for

for epoch in range(epochs):
    for i in range((n - 1) // bs + 1):
        #         set_trace()
        start_i = i * bs
        end_i = start_i + bs
        xb = x_train[start_i:end_i]
        yb = y_train[start_i:end_i]
        pred = model(xb)
        loss = loss_func(pred, yb)

        loss.backward()
        with torch.no_grad():
            weights -= weights.grad * lr
            bias -= bias.grad * lr
            weights.grad.zero_()
            bias.grad.zero_()
```

Training loop - forward path

Neuron activation gradient

$$z(\mathbf{w}, b, \mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + b$$

$$activation(z) = max(0, z)$$

The vector chain rule tells us:

$$\frac{\partial activation}{\partial \mathbf{w}} = \frac{\partial activation}{\partial z} \frac{\partial z}{\partial \mathbf{w}}$$

$$\frac{\partial activation}{\partial \mathbf{w}} = \begin{cases} \vec{0}^T & \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ \mathbf{x}^T & \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases}$$

$$\frac{\partial activation}{\partial b} = \begin{cases} 0 \frac{\partial z}{\partial b} = 0 & \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ 1 \frac{\partial z}{\partial b} = 1 & \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases}$$

What it means?

- Partial derivative of every neuron activation function with respect to optimised parameter (w, b)
- Summary of all sensitivities along every epoch of optimisation (Jacobian matrix)
- Possible identification of different effects by varying activation functions
- Gradients as inputs for optimising the loss function

Loss functions

Mean squared error

$$C(\mathbf{w}, b, X, \mathbf{y}) = \frac{1}{N}\sum_{i=1}^{N}(y_i - activation(\mathbf{x}_i))^2 = \frac{1}{N}\sum_{i=1}^{N}(y_i - max(0, \mathbf{w}\cdot\mathbf{x}_i + b))^2$$

...Cross Entropy - multi categorical

- Space of possible solutions maps onto a smooth range (maximum likelihood)
- Measuring the error between two probability distributions
- Penalising confident predictions which are wrong (false positives)

...Wasserstein GAN

Back Propagation

- Loss derivative calculation
- Optimisers (e.g. SGD, Momentum, Adam)
- Learning rate hyperparameter

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \boxed{\frac{\partial C}{\partial \mathbf{w}}}$$

- Chain rule of loss function

$$\begin{aligned} u(\mathbf{w}, b, \mathbf{x}) &= max(0, \mathbf{w} \cdot \mathbf{x} + b) \\ v(y, u) &= y - u \\ C(v) &= \frac{1}{N} \sum_{i=1}^{N} v^2 \end{aligned}$$

```python
lr = 0.5  # learning rate
epochs = 2  # how many epochs to train for

for epoch in range(epochs):
    for i in range((n - 1) // bs + 1):
        #         set_trace()
        start_i = i * bs
        end_i = start_i + bs
        xb = x_train[start_i:end_i]
        yb = y_train[start_i:end_i]
        pred = model(xb)
        loss = loss_func(pred, yb)

        loss.backward()
        with torch.no_grad():
            weights -= weights.grad * lr
            bias -= bias.grad * lr
            weights.grad.zero_()
            bias.grad.zero_()
```

Gradient with respect to the weights

From before, we know:

$$\frac{\partial}{\partial \mathbf{w}} u(\mathbf{w}, b, \mathbf{x}) = \begin{cases} \vec{0}^T & \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ \mathbf{x}^T & \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases}$$

and

$$\frac{\partial v(y, u)}{\partial \mathbf{w}} = \frac{\partial}{\partial \mathbf{w}}(y - u) = \vec{0}^T - \frac{\partial u}{\partial \mathbf{w}} = -\frac{\partial u}{\partial \mathbf{w}} = \begin{cases} \vec{0}^T & \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ -\mathbf{x}^T & \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases}$$

https://arxiv.org/abs/1802.01528

Gradient with respect to the weights

$$\frac{\partial C(v)}{\partial \mathbf{w}} = \begin{cases} \vec{0}^T & \mathbf{w} \cdot \mathbf{x}_i + b \leq 0 \\ \frac{2}{N} \sum_{i=1}^{N} (\mathbf{w} \cdot \mathbf{x}_i + b - y_i)\mathbf{x}_i^T & \mathbf{w} \cdot \mathbf{x}_i + b > 0 \end{cases}$$

Interpretation:

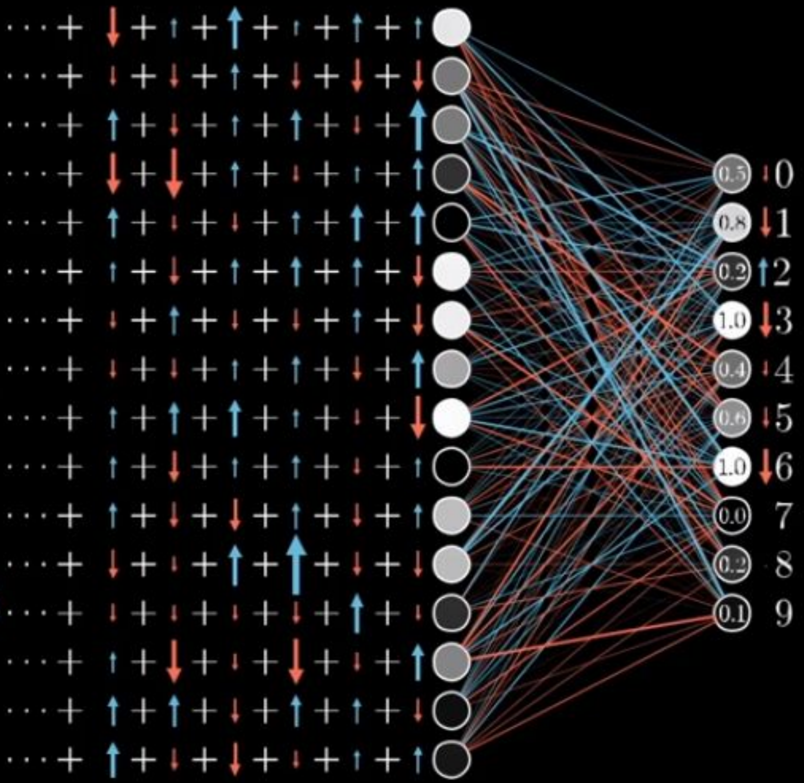
- Gradients as weighted average over all **X** input features
- Error term as weighting factor and directional indication of optimisation
- Scaling factor of gradient descent 2/N

Propagate backwards

Increase b

Increase w_i
in proportion to a_i

Change a_i
in proportion to w_i



Training Loop - summary

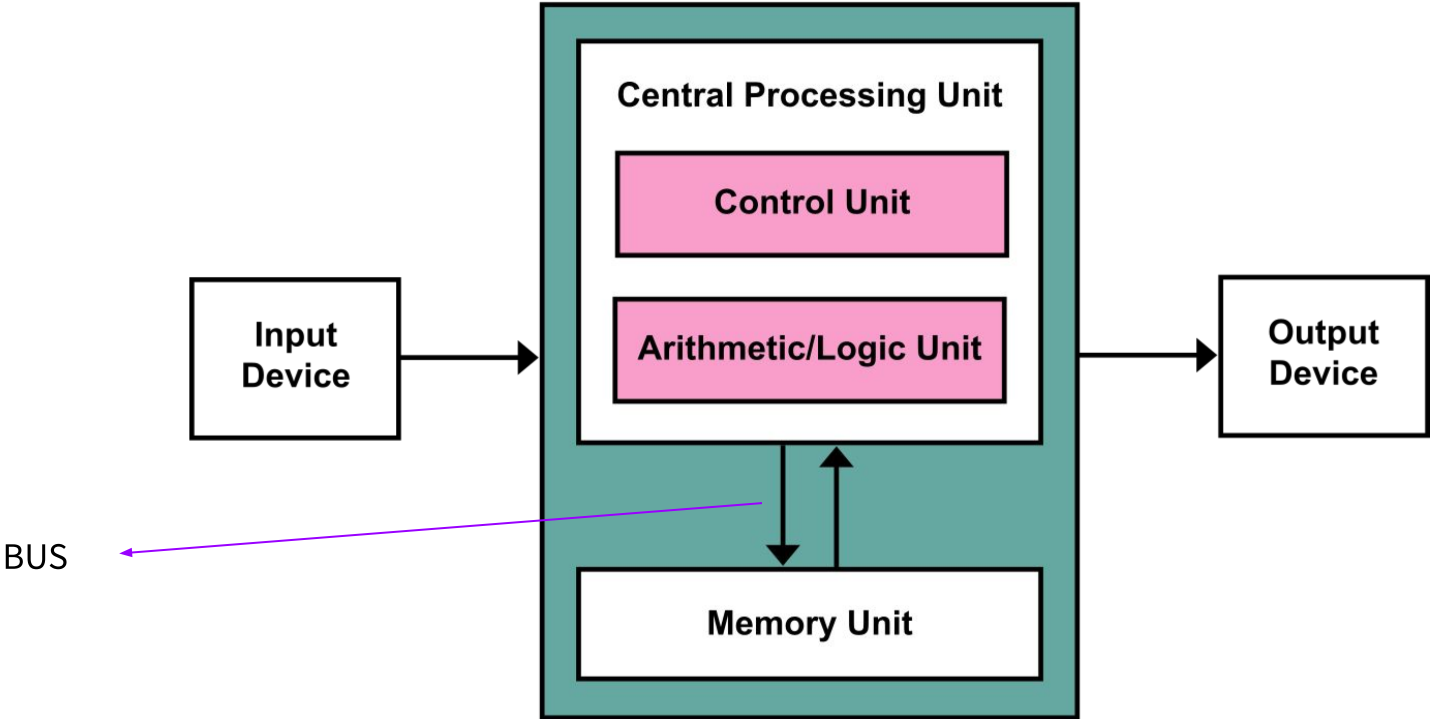
```
lr = 0.5  # learning rate
epochs = 2  # how many epochs to train for

for epoch in range(epochs):
    for i in range((n - 1) // bs + 1):
        #         set_trace()
        start_i = i * bs
        end_i = start_i + bs
        xb = x_train[start_i:end_i]
        yb = y_train[start_i:end_i]
        pred = model(xb)
        loss = loss_func(pred, yb)

        loss.backward()
        with torch.no_grad():
            weights -= weights.grad * lr
            bias -= bias.grad * lr
            weights.grad.zero_()
            bias.grad.zero_()
```

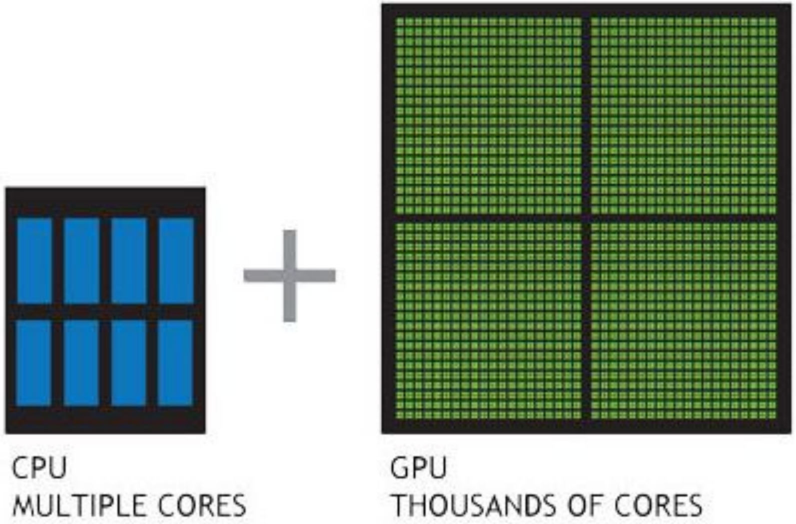
Why vectors are faster?

Why vectors?



Central Processing Unit
Control Unit
Arithmetic/Logic Unit
Input Device
Output Device
Memory Unit
BUS

Why Vectors?



Python - dunder methods

__init__

__call__

Iterations:

__len__

__getitem__

Context Manage (with …):

__enter__

__exit__

Python decorators

Syntactic shortcut of Function composition:

```
def f():
    pass
f=my_decorator(f)
```

Python decorators

```
from functools import wraps, partial

def decorator(func=None, parameter1=None, parameter2=None):
   if not func:
        # The only drawback is that for functions there is no thing
        # like "self" - we have to rely on the decorator
        # function name on the module namespace
        return partial(decorator, parameter1=parameter1, parameter2=parameter2)
   @wraps(func)
   def wrapper(*args, **kwargs):
        # Decorator code-  parameter1, etc... can be used
        # freely here
        return func(*args, **kwargs)
   return wrapper
```

Python decorators

```
@decorator
def my_func():
    pass

@decorator(parameter1="example.com", ...):
def my_func():
    pass
```

python 3.8 supports positional only parameters:

```
def decorator(func=None, /, parameter1=None, parameter2=None, *):
```

https://docs.python.org/3/whatsnew/3.8.html#positional-only-parameters

Best Practices

- Aspire for ‘Flatten’ code - don’t indent too much
  - Conditions that immediately return should be positioned first
- Functions should receive no more than 2-3 parameters
  - Understand the business - know where to cut
  - Consider passing object(s) instead
- A function should do only one thing (!)
  - Easier to test
  - Less bugs
  - Reusability

TorchScript

Tracing a model execution with parameters:

`torch.jit.trace`

Compiling a model to TorchScript:

`Torch.jit.script`

Loading a saved TorchScript:

`Torch.jit.load`

Faster execution in production &

Scripts can be loaded also in C++ (!)

https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html

So, what did we learn?