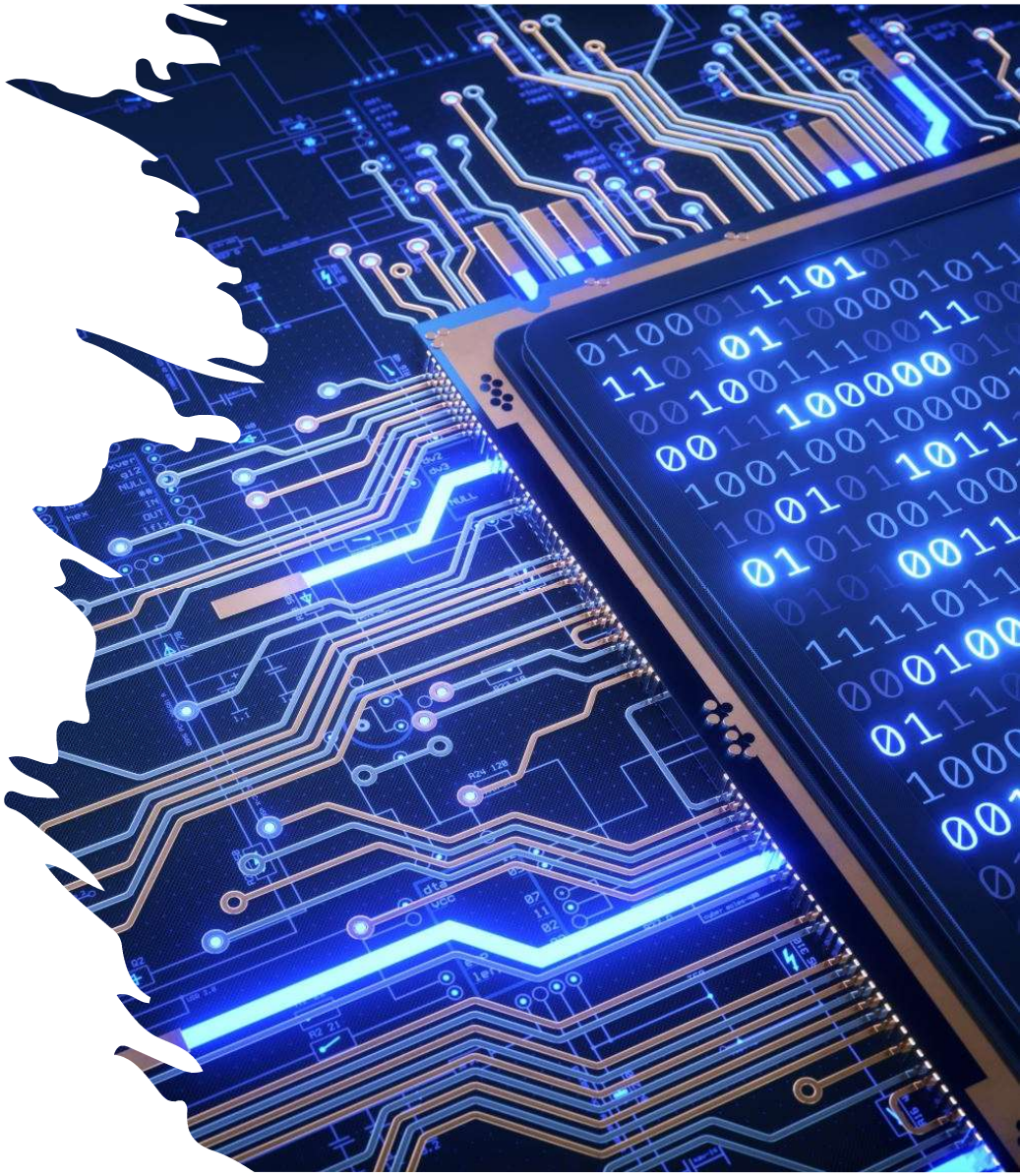
Deep Learning

Week 2:

Loss functions, computational graph, pytorch intro



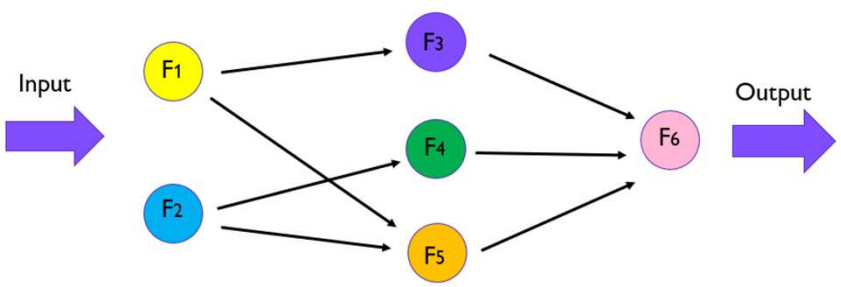
Last week

Neural networks:

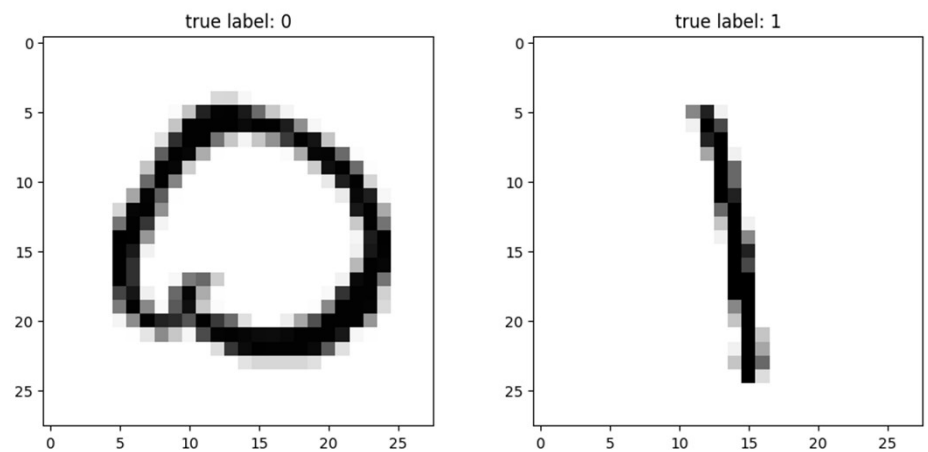
- Invented in the 1950'ies

- Large collection of simple units performing *nonlinear* operations.

- Logistic regression is a simple neural network



Logistic regression:
In the exercise, you implemented a 'single-layer perceptron' for binary classification, using numpy.



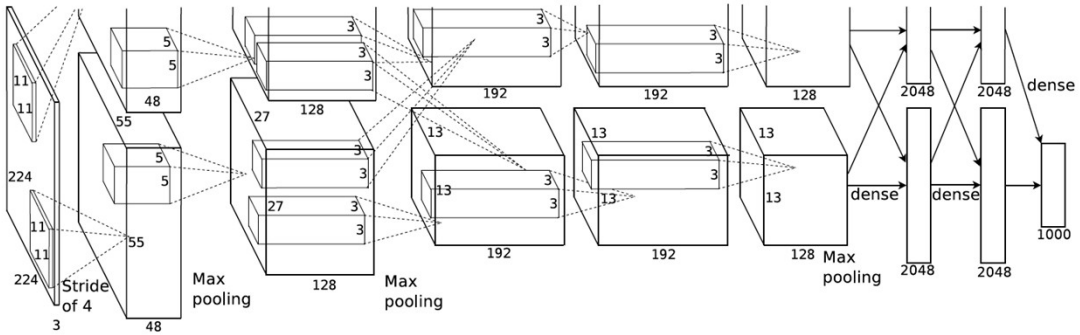
Quiz!

What was special about AlexNet?

A: It was really deep

B: It used graphical processing units

C: It was Geoffrey Hinton's first paper



Topics we will discuss today:

- Loss functions
- Gradient descent
- The computational graph
- Pytorch: PyTorch

Loss landscapes

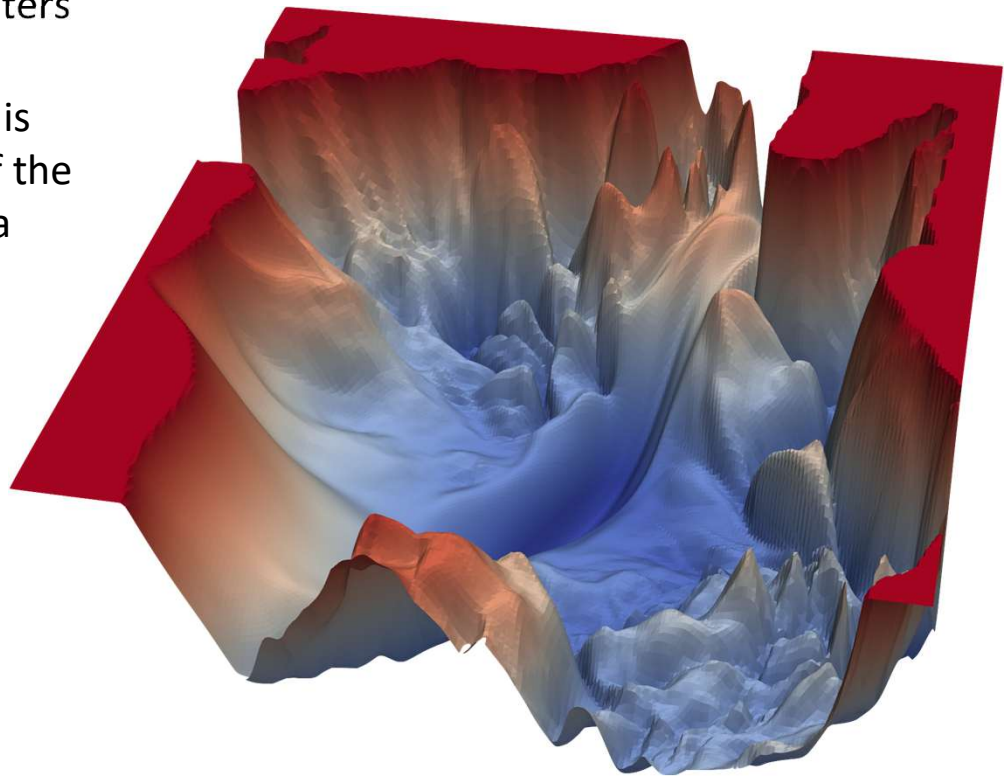
We optimize neural networks by minimizing a 'cost' or a 'loss':

$$L(y, \hat{y}), \hat{y} = f(\theta, X), \qquad \theta\text{: model parameters}$$

The loss tells the network whether a given output ($\hat{y}$) is considered close to the target (y), and the gradient of the loss with respect to θ is used to update the model in a more favourable direction.

Following the negative gradient 'down hill' is called *gradient descent*.

Think of the loss as a landscape:



However, L is defined for a single data point

What do we do when we have 1e5 data points to train on?

1. Calculate 1e5 gradients, and update all parameters 1e5 times?
   (1e5 gradients -> 1e5 updates in a loop)

2. Calculate the total loss, get a scalar, and update all parameters once?

3. Calculate the loss for each data point, and update the parameters immediately?
   (1 gradient -> 1 update, 1e5 times)

4. Something else?

1. Calculate 1e5 gradients, and apply all 1e5 updates together?

This is a huge amount of calculations (imagine a model with 1e6 parameters), and only the first update would actually be along the gradient.

2. Calculate the total loss, getting a scalar, and updating all parameters once?

This could work, but getting through the whole data set can be slow, and there can be some issues with convergence for deeply non-convex loss landscapes.

3. Calculate the loss for each datapoint, and update the parameters immediately?

This could also work, but it's incredibly noisy - the gradient of a single datapoint is not a good approximation for the gradient of the whole data set.

4: Minibatches to the rescue:

We take a subset of data points (N = 32, 64 and 128 are popular choices), and calculate the total loss over this:

$$L_{mb} = \sum_{minibatch} loss(y, \hat{y})$$

For large N, $\frac{\partial L_{mb}}{\partial \theta_i}$ can still be a decent estimate for $\frac{\partial L_{tot}}{\partial \theta_i}$, while requiring less computations. For intermediate N, the variation in the gradient estimate can have the useful property that the network can better escape local minima (this assumes that deep minima are also wide, and thus harder to escape).

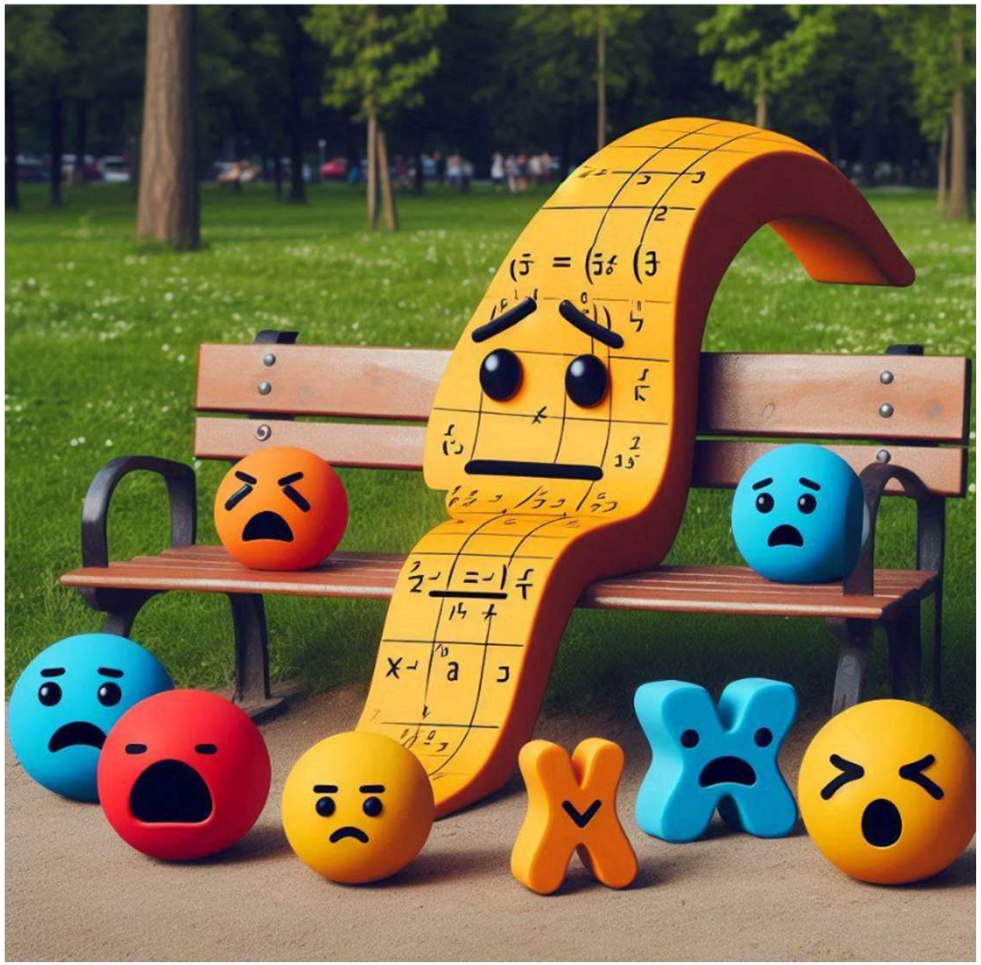
Because of the added randomness, gradient descent with minibatches is called *stochastic gradient descent (SGD)*.

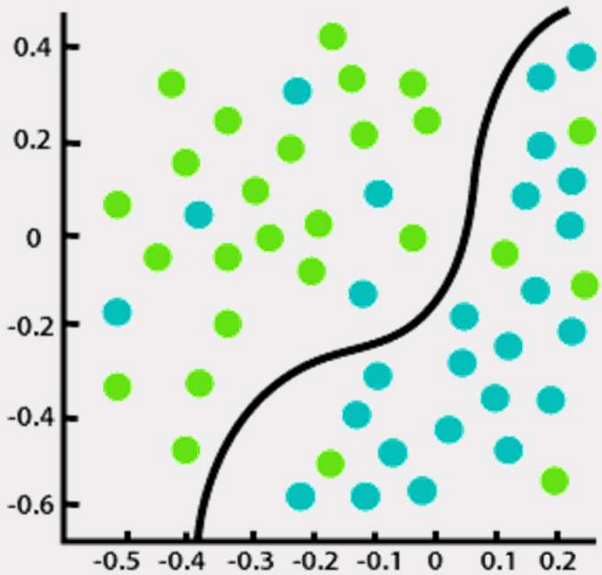
Common practice is to draw the minibatches 'without replacement', to ensure equal coverage of the whole training set. Each full evaluation of the training set is called an '**epoch**'.

It is a good idea to ensure that the training set is shuffled between epochs, such that the data is not 'batched' in the same way twice.

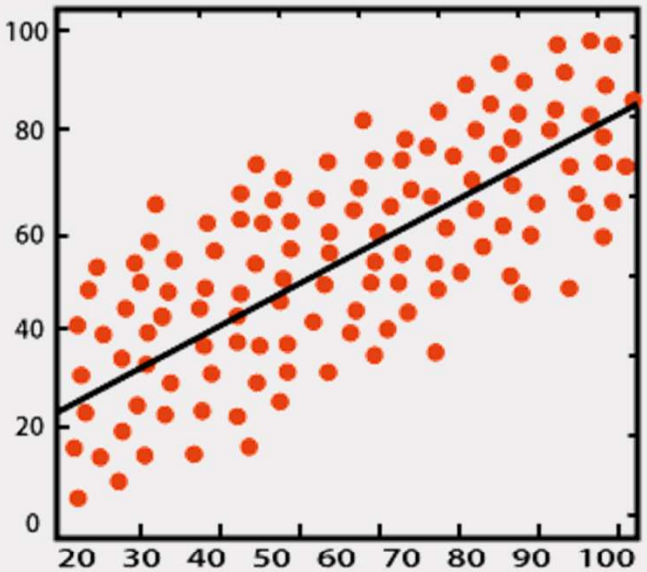
Loss functions

What Bing Chat thinks a loss function looks like:



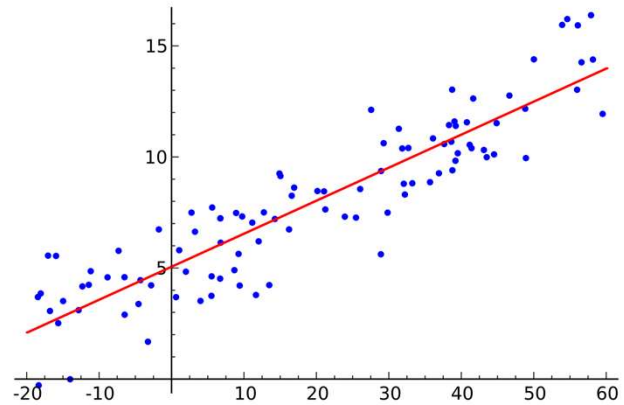


Classification



Regression

Regression loss functions



In regression, the goal is to get as close as possible to a set of recordings.

In **least squares regression**, L is usually the sum of squared residuals:

$$L(y, \hat{y}) = \sum (y_i - \hat{y}_i)^2$$

In general, the p'th norm of the residuals can work as a loss function:

$$L_p(y, \hat{y}) = |y - \hat{y}|_p = (\sum |y - \hat{y}_i|^p)^{\frac{1}{p}}$$

(so, larger p means greater penalty for large deviations)

More elaborate examples include 'smoothed L1 loss', where L behaves as a 1-norm for large deviations and as L2 for small deviations ('Huber' loss).

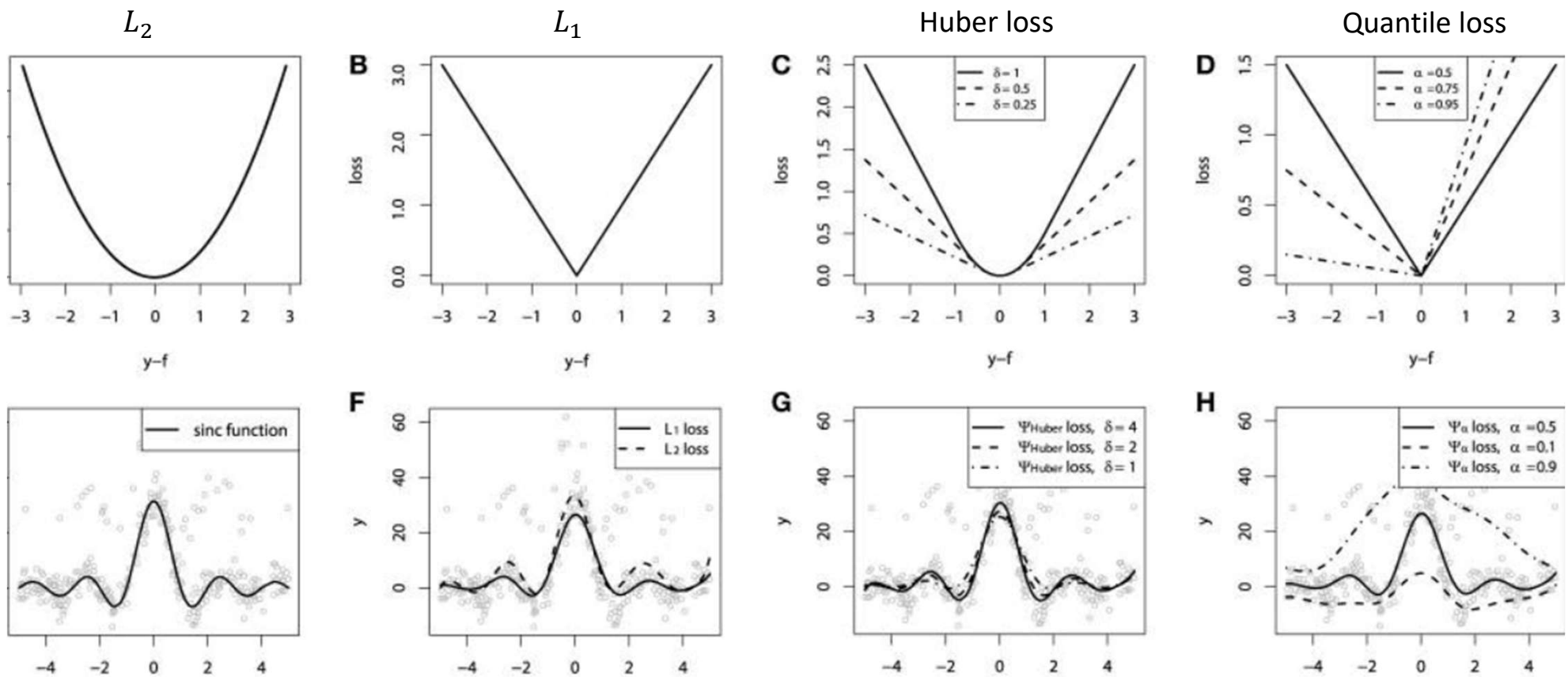
Even more elaborate is the *quantile loss function*, which attempts to predict the q'th quantile of a distribution of observations:

$$L_q(y, \hat{y}) = (1 - q) \sum_{y_i < \hat{y}_i} |y_i - \hat{y}_i| + q \sum_{y_i > \hat{y}_i} |y_i - \hat{y}_i|$$

(the logic is that overshooting and undershooting y have different penalties)

Please note that while L functions described here are convex in $\hat{y}$, they can not be assumed to be so in θ-space, which is what matters.

1D losses:



Source: doi:[10.3389/fnbot.2013.00021](https://doi.org/10.3389/fnbot.2013.00021)

Classification

Classification loss functions

The loss functions for regression work poorly for classification – if, for instance, the correct class is '1', we should not treat '2' as being more correct than '5'.

Instead, let us consider probabilities instead of classes, in particular likelihood functions:

$$Likelihood\ for\ whole\ minibatch = \prod_{i}^{N} \hat{P}(c_i|X_i,\theta) = \prod_{i}^{N} f(c_i|X_i,\theta)$$

($\hat{P}(c_i|X_i,\theta)$ is the estimated probability of the correct label being the label, for input i)

This is the likelihood of seeing the actual data (with actual labels c_i), as estimated by the given model.
Ideally, our model should assign a large likelihood to the training data.
Multiplying a lot of potentially very small numbers is numerically unstable, so we prefer the log likelihood:

$$LL = \sum_{i} \log \hat{P}\ (c_i\ |\theta, X_i)$$

And by changing sign, we get a loss which we can minimize:

$$L = -\sum_{i} \log \hat{P}\ (c_i\ |\theta, X_i)$$

(this is the 'negative log likelihood loss', and the sum is over a minibatch)

Classification loss functions

We can also reach this equation from another direction:

The differences between discrete probability distributions q_j and p_j ($j = 1, \dots, n_{classes}$) can be described using the cross entropy:

$$H(p,q) = -\sum_j p_j \log(q_j)$$

If we let $q_j = \hat{P}(c_i|X_i,\theta)$ (estimated probability of class j for input X and parameters θ), and

$$p_j = \begin{cases} 1 \text{ for } j = c \\ 0 \text{ else} \end{cases}$$

(1 for the true class, 0 otherwise)

Then $H(p,q)$ is the distance between the true distribution and the estimated one, depending on θ, and we could sum that

$$\mathrm{H(p,q)} = -\log(\mathrm{q_j}) \Rightarrow L = -\sum_i \log \hat{P}(c_i\,|\theta, X_i)$$

Because of this, the *negative log likelihood loss* is also called the *cross entropy loss*.

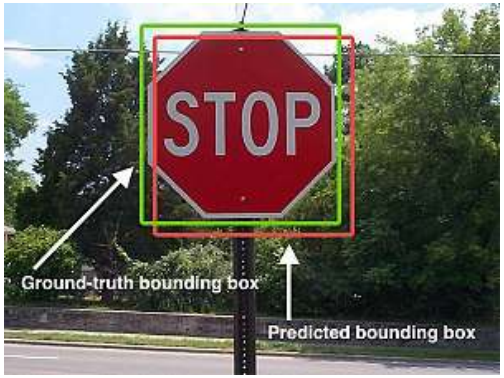
The logistic regression loss from last week was in fact a negative log likelihood loss for two possible classes.

Dice loss / Jaccard index / intersection-over-union:

When doing highly imbalanced classification, such as object detection in images, frequently the (negative) intersection-over-union (IoU) is used.

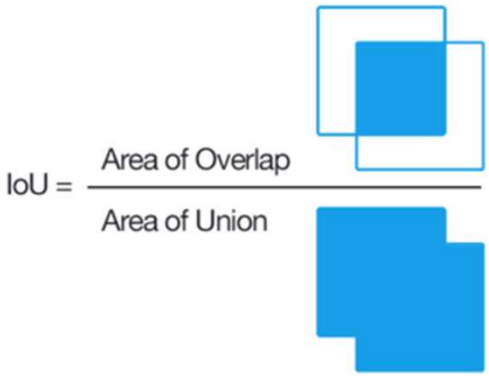
It is also known as the Jaccard index, described as:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$



For semantic segmentation (pixel-wise classification), the 'Dice loss' is yet another implementation of this:

$$D(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -2\frac{\sum_i y_i \hat{y}_i}{\sum_i \hat{y}_i^2 + \sum_i {y_i}^2}$$



(y_i is the vector describing whether the object is in pixel *i*)

Categorial encoding

We have not yet discussed *how* the network should output $\hat{P}\left(c_j | X_i, \theta\right)$:

(blackboard drawing)

Let $\{X_i, l_i\}_i$ be a labeled dataset, $l = 1, \dots, n_l$ (n_l classes). Prior to training, we *one-hot-encode* the labels:

$$l_i = 3 \rightarrow [0,0,1,0, \dots, 0]$$

(the one-hot-encoded vector is n_l long, with 1 in the l_i'th place)

This means that the *i*'th entry in the encoded vector is the probability of the label being *i*. Simultaneously, we make sure that the network outputs an n_l-dimensional vector, of values between 0 and 1.

We do this using a *softmax layer*.

Softmax

Generally, the output of a neural network may not be restricted to the interval (0,1), which is necessary for the probability interpretation. It is then costumary to scale the outputs for any given input(i), using a *softmax* function:

$$SM(v) = \frac{e^{v_c}}{\sum_j e^{v_j}}, j, c\text{: output classes}$$

The softmax can be the last layer of the network, or it can be built into the loss function.

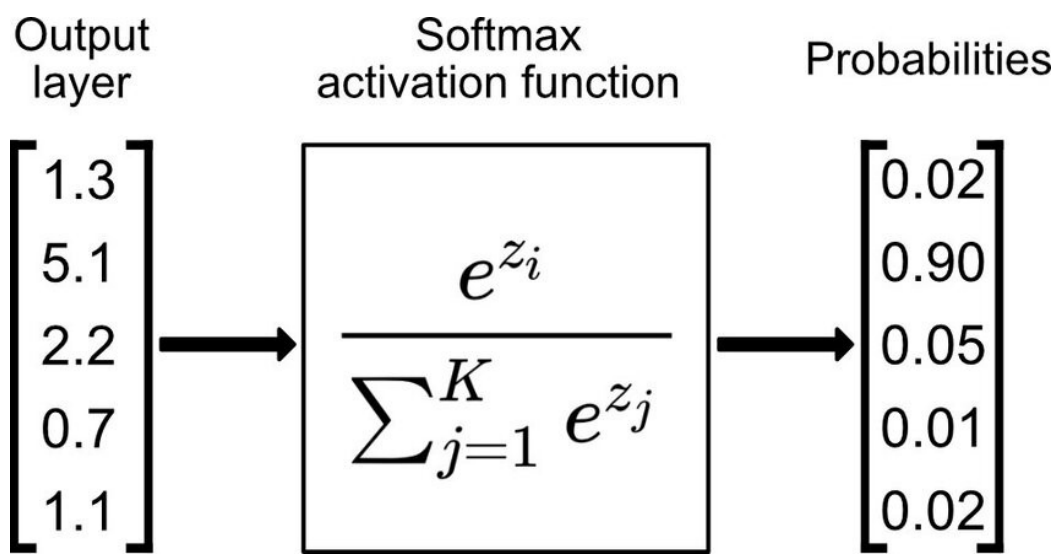
Relevant reading: https://pytorch.org/docs/stable/generated/torch.nn.Softmax.html

Softmax interpretation:

The outputs from a softmax layer sum to 1, and they are all between 0 and 1.

So can we treat them as probabilities?

Please discuss with your neighbour



Answer: *Sort of*

Proof for 'yes':

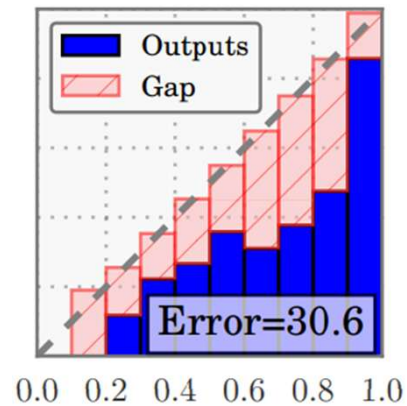
10.1162/neco.1996.8.1.41 (can be downloaded from course page)

Proof is only valid for 'perfectly trained' networks with sufficient flexibility.

In reality, not always:

https://proceedings.mlr.press/v70/guo17a/guo17a.pdf

(they find that even large and well-trained networks have a tendency towards overconfidence)



Taken from Guo et al.

In general, this topic is an example of *uncertainty quantification*:

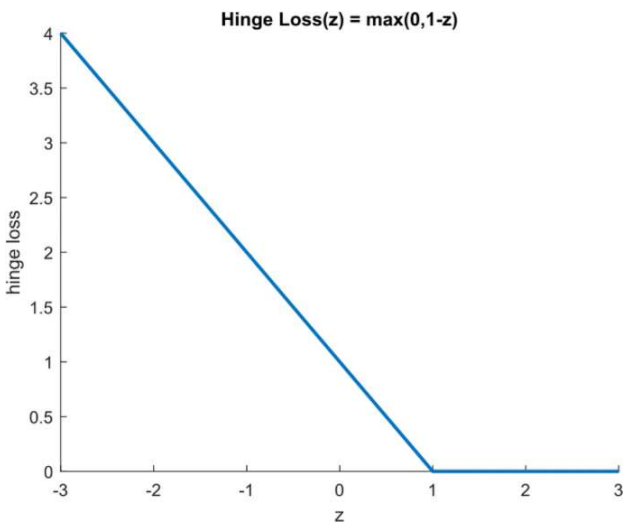
https://www.sciencedirect.com/science/article/pii/S1566253521001081

Hinge loss

Another noteworthy classification loss is the *hinge loss*, which is essentially what is used in support vector machines:

$$L_i = \sum_{c_j \neq c_i} \max(0, \Delta + f\left(c_j | X_i, \theta\right) - f(c_i | X_i, \theta))$$

(if the amplitude of c_i is much larger than for the other classes, we stop caring and set its term to 0)



Please note that in the above, we do not assume that outputs have been scaled in any way. Δ is the 'margin', and is usually set to 1 (in this case, there is no softmax layer).

Regularization

A common trick to reduce overfitting, is to force the network to use as few of its parameters as possible – we reward it for settings parameters to 0:

$$L_{tot}(y, \hat{y}) = L(y, \hat{y}) + L_{reg}(\theta)\ ,\ \theta\text{: network parameters}$$

L_{reg} is most commonly the L1 or L2 norm (or their combination, the '*elastic net*'):

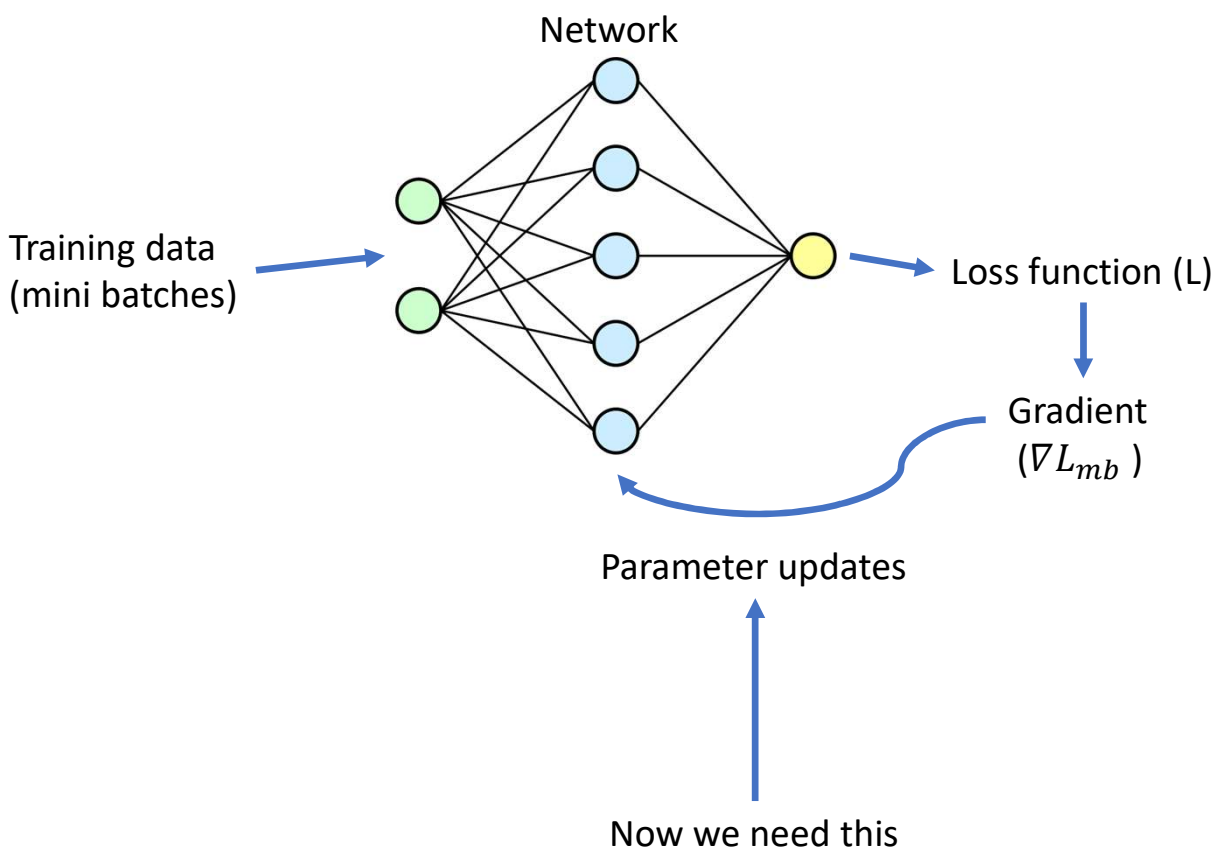
$$||\theta||_1 = \frac{1}{n}\sum_{i=1}^{n}|\theta_i|\ ,\ ||\theta||_2 = \frac{1}{n}\sum_{i=1}^{n}|\theta_i|^2\ ,\ \big|||\theta||_{EN} = \alpha\big||\theta||_1 + \beta||\theta||_2$$

To implement these in pytorch, you can use the 'weight_decay' parameter in the optimizer for L2, and the following for L1:

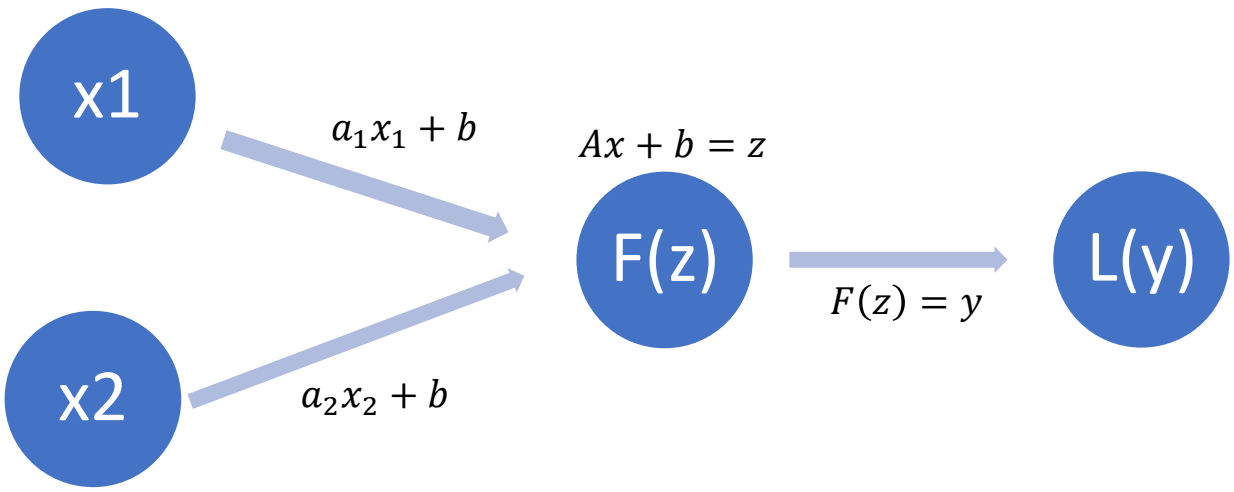
```
l1 = nn.L1Loss(size_average=False)
l1_loss = 0
for param in model.parameters():
   l1_loss += l1(param)
```

Break

Status so far:



Backpropagation on the computational graph



Given this, how do we use SGD to update a_1?

$$a_1 \rightarrow a_1 - \delta \frac{\partial L}{\partial a_1}$$

By the chain rule:

$$\frac{\partial L}{\partial a_1} = \frac{\partial L}{\partial y}\frac{\partial F}{\partial z}\frac{\partial z}{\partial a_1}$$

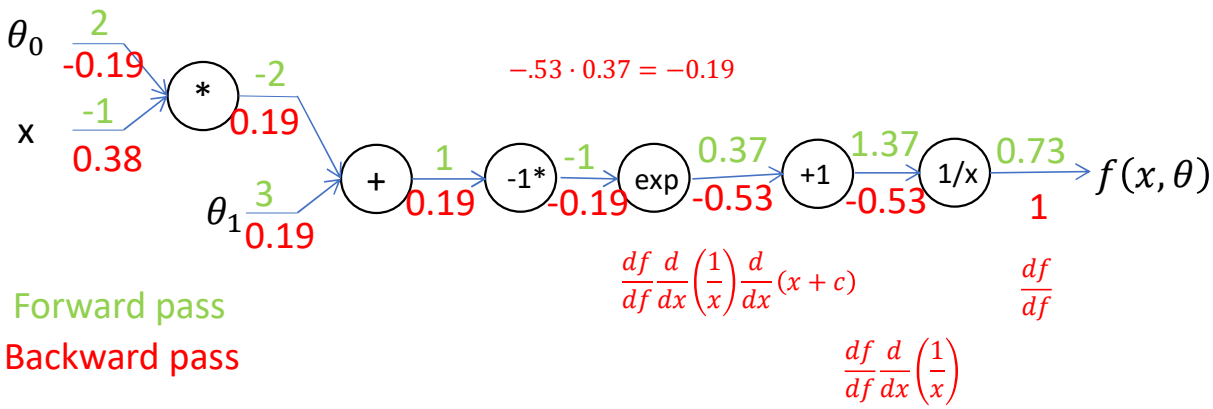
Let's assume we are doing regression with an L1-norm and let $F(z) = \max(0, z)$. Let also x1=1, $\frac{\partial L}{\partial y} = 1, z > 0$:

(blackboard)

Backpropagation on the computational graph

Numerical example:

Let us write out the computational graph for $f(x,\theta) = \frac{1}{1+e^{-(\theta_0 x+\theta_1)}}$:



$$\frac{d}{dx}\left(\frac{1}{x}\right) = -\frac{1}{x^2} \qquad \frac{d}{dx}(c+x) = 1$$

$$\frac{d}{dx}(e^x) = e^x \qquad \frac{d}{dx}(ax) = a$$

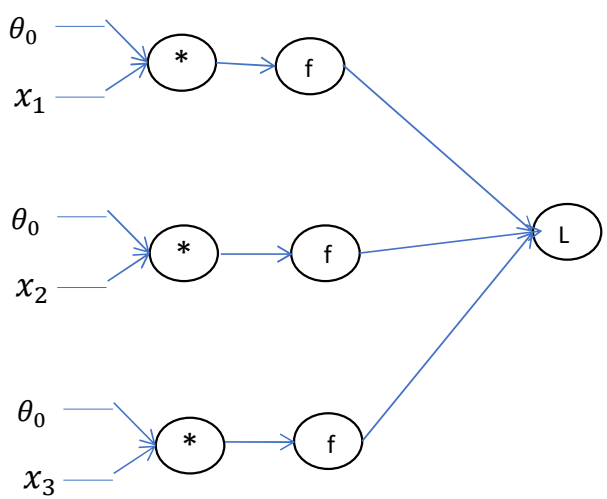
$$\frac{d}{dx}(e^x) = e^x$$

Let x=-1, $\theta_0 = 2, \theta_1 = 3$

Question: what is $\frac{df}{d\theta_0}, \frac{df}{d\theta_1}$?

Minibatches & the graph

Considering the computational graph together with minibatches shows us why the loss must be reduced to a scalar before backpropagation:



Remember that it's the same θ_0 in all three branches, so

$$\frac{\partial L}{\partial \theta_0} = \frac{\partial}{\partial \theta_0}\left[\frac{1}{3}\sum_{i=1}^{3} f(\theta_0 x_i)\right] = \frac{1}{3}\sum_{i=1}^{3}\frac{\partial}{\partial \theta_0}[f(\theta_0 x_i)]$$

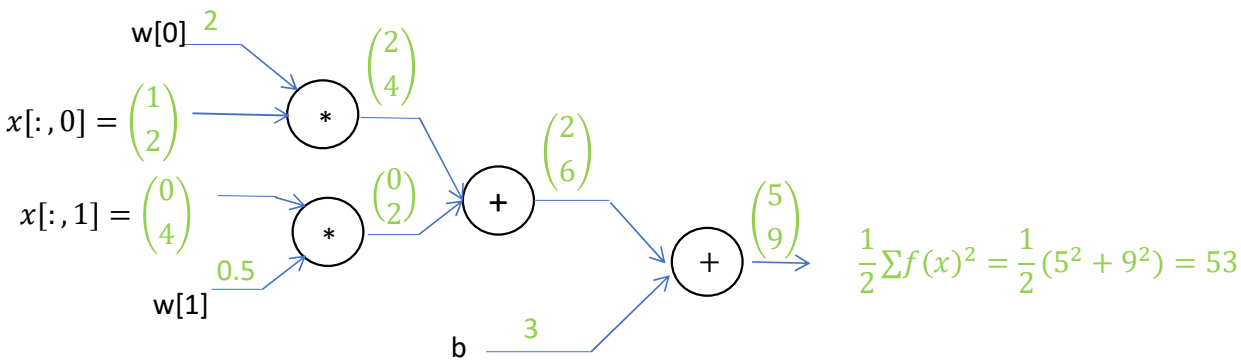
This way, we can get the average gradient by starting from the average loss. ($< \frac{dL}{d\theta} > = \frac{d<L>}{d\theta}$)

Class exercise:

Let us consider the function $f(x, w, b) = x \cdot w + b$ for $x \in R^2$, and assume we have a minibatch of size 2, meaning that X has size 2 x 2 (batch X data-dim). Further, let us assume that

$L = \frac{1}{batchSize} \sum_{batch} f(x[iBatch, :])^2 = \frac{1}{2}(f(x[0, :])^2 + f(x[1, :])^2)$. Let $X = \begin{vmatrix} 1 & 0 \\ 2 & 4 \end{vmatrix}$, w=[2,0.5], b=3

We can draw this situation as:



$\frac{1}{2} \sum f(x)^2 = \frac{1}{2}(5^2 + 9^2) = 53$

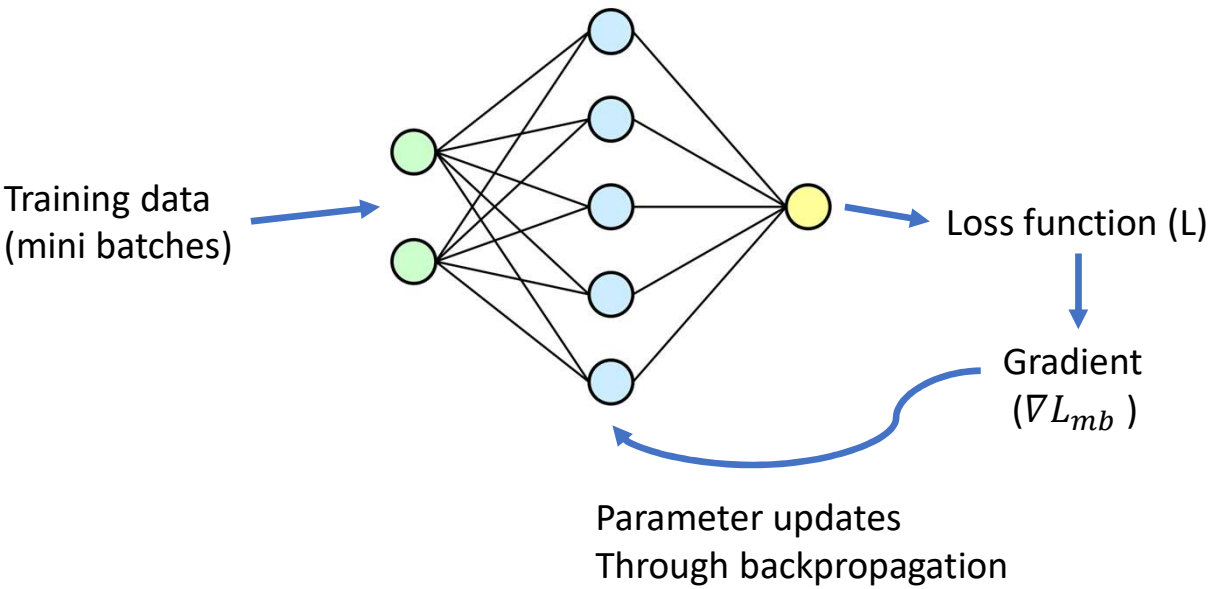
Question: what is $\frac{\partial L}{\partial w[1]}$ in this minibatch?

$$\frac{\partial}{\partial \theta} \frac{1}{n} \sum_i f(x_i, \theta) = \sum_i \left[\frac{\partial}{\partial \theta} \frac{1}{n} f(x_i, \theta) \right]$$

```python
import torch

x=torch.tensor([1.0,2,0,4],requires_grad=True).reshape(2,2).T
w=torch.tensor([2,.5],requires_grad=True)
b=torch.tensor(3.0,requires_grad=True)
L=torch.mean((torch.matmul(x,w)+b)**2)
print(L)
L.backward()
print(w.grad)
print(b.grad)
```

```
tensor(53., grad_fn=<MeanBackward0>)
tensor([23., 36.])
tensor(14.)
```

Recap



$$(\theta \rightarrow \theta - \delta \frac{\partial L}{\partial \theta})$$

Activation functions

Activation functions

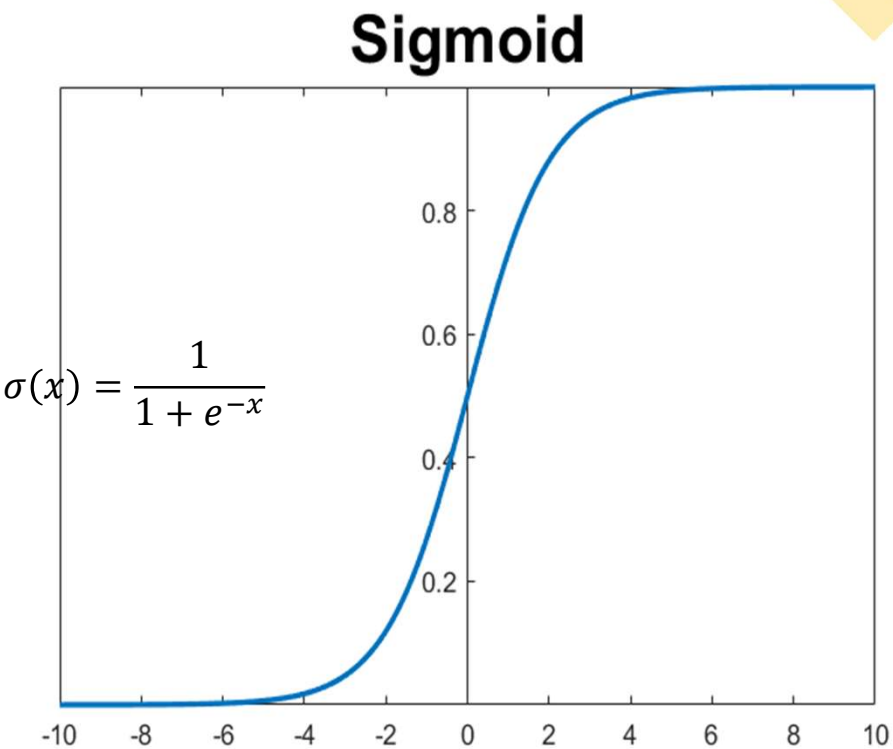
We already hinted that the ‘neurons’ in a neural network are nonlinear functions. These are generally called ‘activation functions’. Let’s talk about some common candidates.

The **sigmoid** is considered the ‘original’ activation function. However, it has short-comings.

Can you come up with some issues?

- Gradient -> 0 for large and small inputs (meaning slow learning).
- It assumes 0-centered input, but does not return 0-centered output.
- Uses exp(), which is somewhat slow

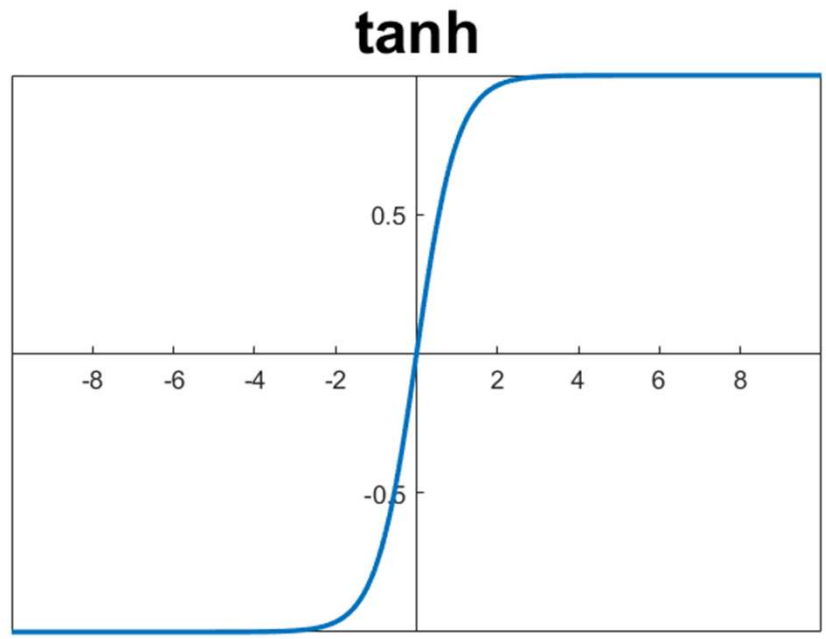
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$



Activation functions

An upgrade to the sigmoid is tanh, because it centers the output.

However, we still have vanishing gradients and it is still a ‘slow’ function to evaluate.

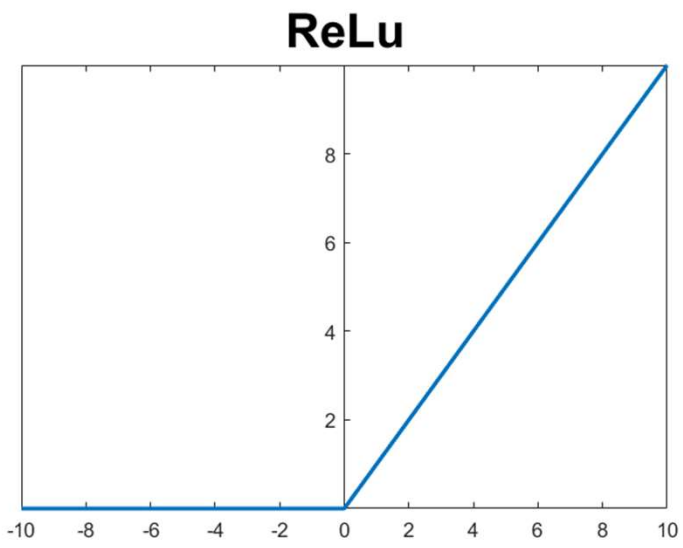


ReLu

A generally favoured activation function is the REctified Linear Unit (ReLu):

It does not saturate for positive inputs, and is quick to evaluate. However:

- It can ‘deactivate’ if all inputs are negative.
- Does not center the outputs



Activation functions

Due to the popularity of the ReLu, a large number of variations have been proposed:

Leaky ReLu

ELU (exponential linear unit)

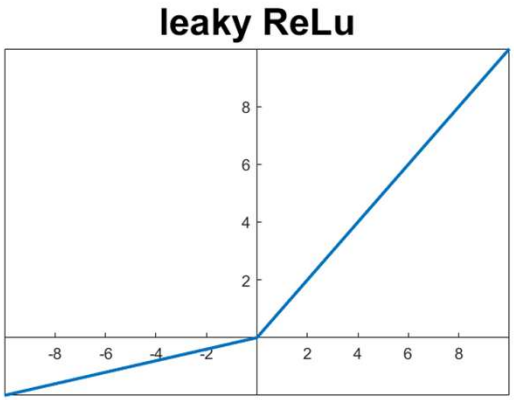
CELU

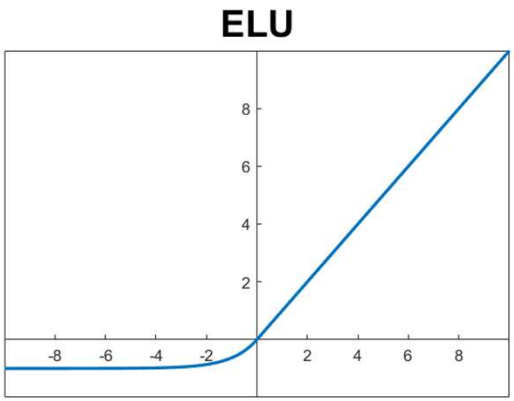
PRELU (Leaky Relu with variable negative-axis slope)

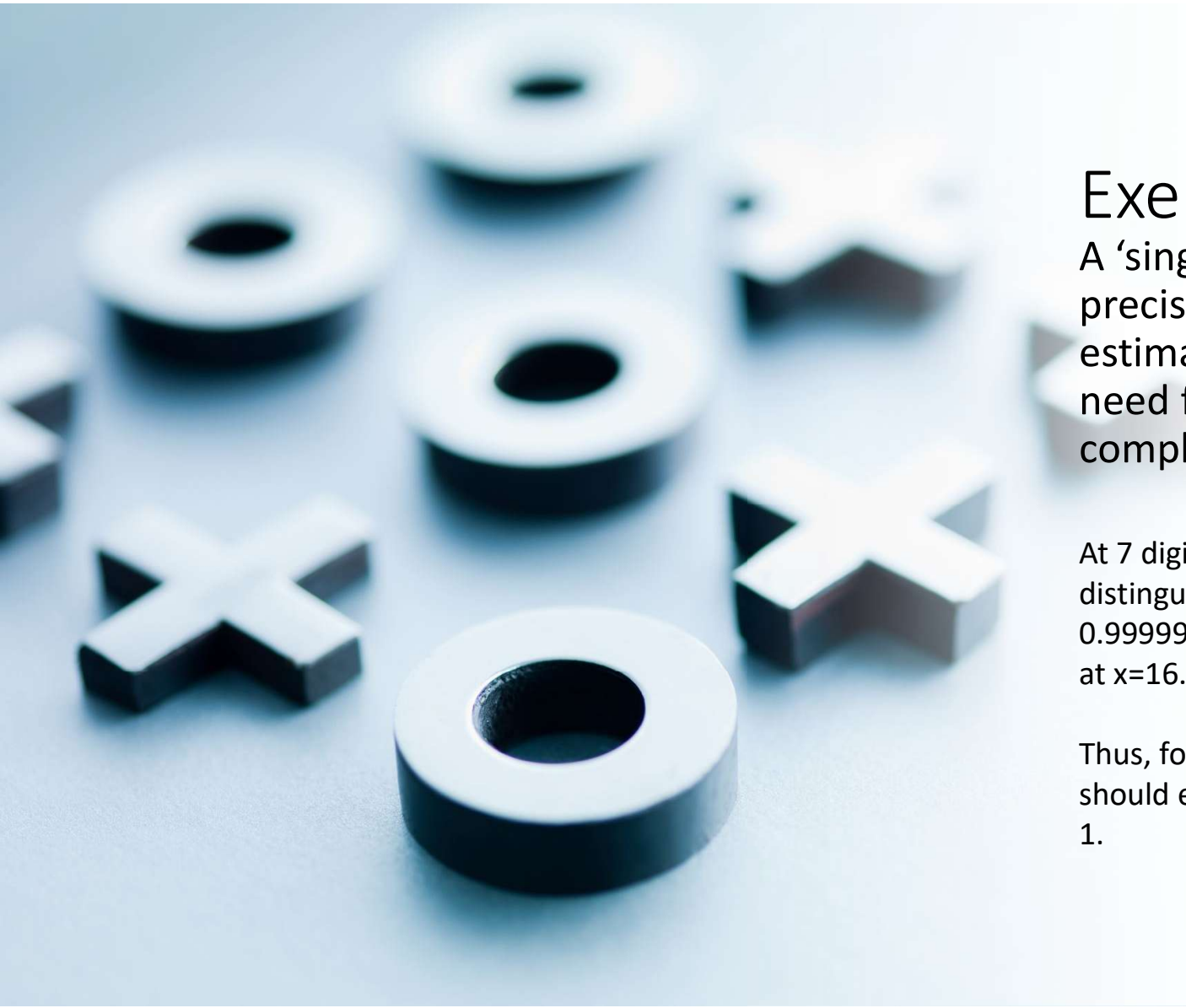
Generally, this advice will get you a long way:

Always use ReLu. If curious, try out Leaky Relu, PRELU or tanh. Inactivated ReLu units can be fixed with batch normalization (future lecture).

Finally, don't use ReLu in the very last layer, unless you really don't need negative output...







Exercise:

A ‘single’ float has around 7 digits of precision. For this data type, estimate how large an input we need for the sigmoid to be completely flat?

At 7 digits of precision, it is hard to distinguish between y and 1.0 after 0.9999999. The sigmoid reaches 0.9999999 at x=16.1.

Thus, for values much beyond 16, we should expect the sigmoid to be constant 1.

Slightly more thorough answer:

```
import math
import numpy as np
eps = np.float32(2**-23)


def sigmoid(x):
  return 1 / (1 + math.exp(-x))

def sigmoid_derivative(x):

    outcome = sigmoid(x)
    return np.float32(outcome * (1 - outcome))


x=0
while(sigmoid_derivative(x)>eps):
    x+=eps

print('After x =',x,'the gradient is than float32 can represent')
```

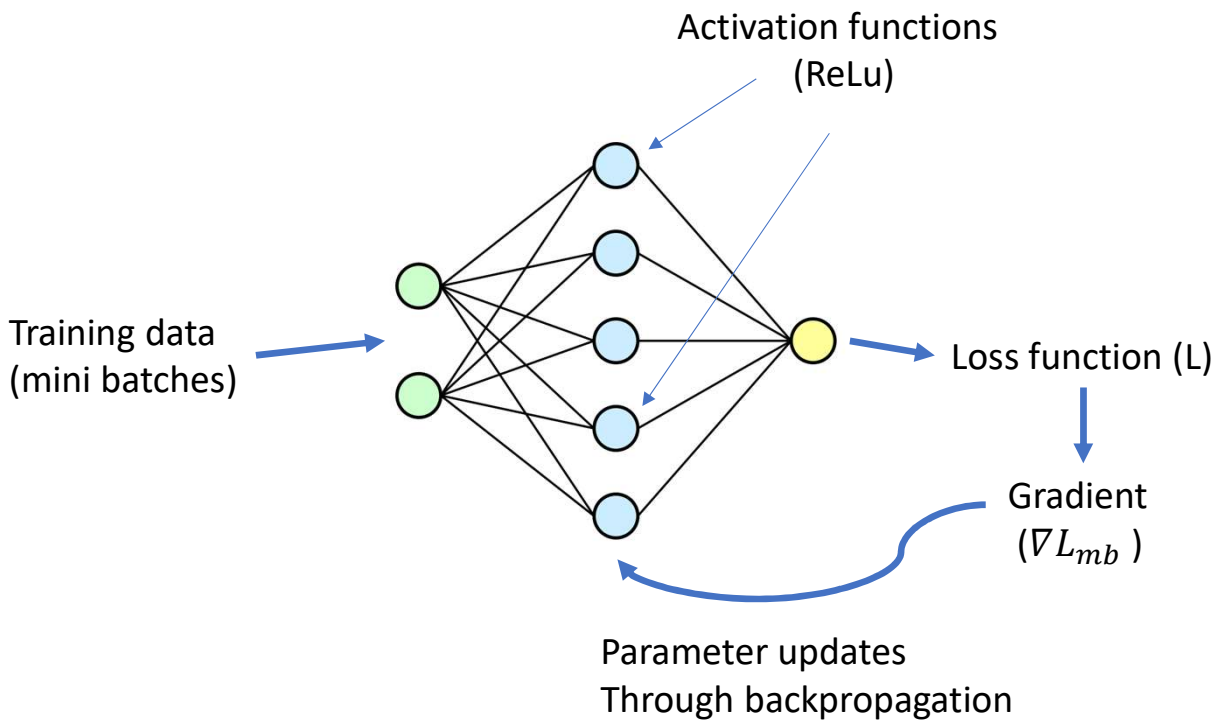
```
After x = 15.942384958267212 the gradient is than float32 can represent
```

There’s a bunch more:

GLU	Logistic
SWIGLU	Softplus
GELU	SELU
Heaviside	SiLU

(if you remember the discussion about universal approx last week, we learned that the group of sufficient activation functions is actually pretty broad)

Recap


Activation functions
(ReLu)
Training data
(mini batches)
Loss function (L)
Gradient
(∇L_{mb})
Parameter updates
Through backpropagation

Things we will get back to in a future lecture:

1. Optimizers (how to update the weights given the gradient)
2. Learning rate schedulers (it need not be constant)
3. Regularization

Break

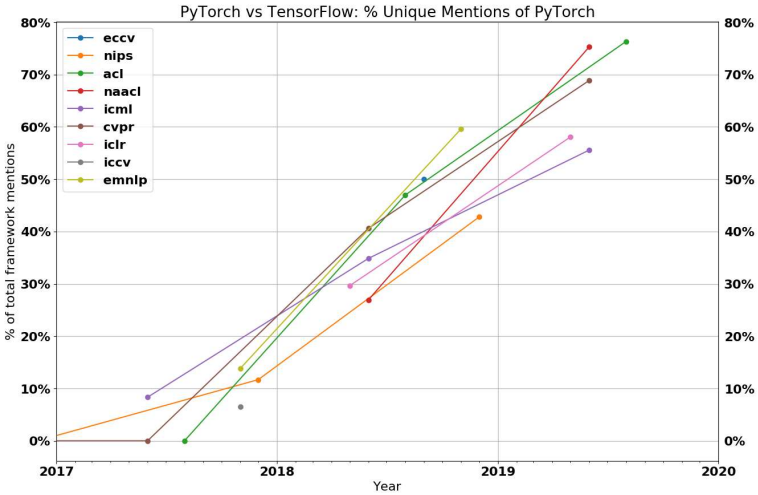
PyTorch

Why pytorch?

PyTorch

or

TensorFlow

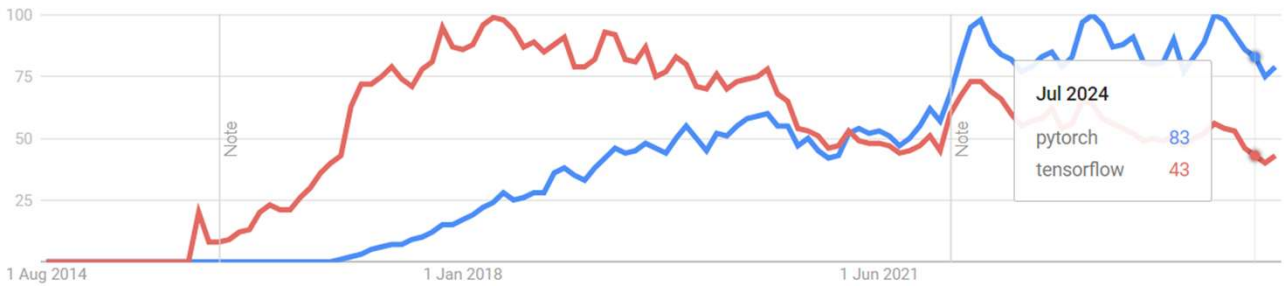


TF and PT are both widespread.

But PT is much more popular in research, due to its flexibility and easier debugging.

TF is considered to have a slight edge in 'production' (or, it did until fairly recently, at least).

Most models on huggingface use PT



2014-2024

Installation

We head to: https://pytorch.org/get-started/locally/ :

PyTorch Build	**Stable (2.8.0)**			Preview (Nightly)	
Your OS	Linux		Mac		**Windows**
Package	**Pip**		LibTorch		**Source**
Language	**Python**			C++ / Java	
Compute Platform	**CUDA 12.6**	CUDA 12.8	CUDA 12.9	~~ROCm 6.4~~	CPU
Run this Command:	`pip3 install torch torchvision --index-url https://download.pytorch.org/whl/cu126`				

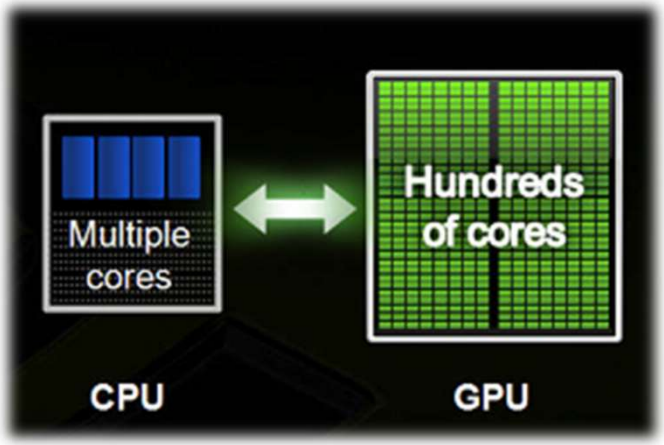
Depending on the settings chosen, we get the command we are supposed to run in a prompt to make the install.

We recommend using Conda as the package manager (but you have to install that first).

Notice that we also have to decide if we want to enable running on a GPU. If so, we need to decide which CUDA driver version to be compatible with. We can download CUDA from [CUDA Toolkit 12.6 Downloads | NVIDIA Developer]
(if you don't have a gpu, just pick 'CPU' option, and forget about CUDA)

GPU's

Each core on a CPU can perform one task at a time (simplified). CPU's in modern laptops may have 4-6 cores. The faster the 'clock' of the cores, the quicker they can perform operations. A typical clock speed may be 2.6-5 GHz.



The same holds true for GPU's. Except that their clock speed is perhaps only 1.9 Ghz. But instead, they can easily have more than 3000 cores.

So, the point of a GPU is that it can do a huge number of small tasks simultaneously. Particularly tasks related to linear algebra. Deep learning is full of small tasks like that.

However, to leverage the GPU for deep learning, we need a way to actually send the deep learning calculations to the GPU. The CUDA library bridges this gap for us. Without a way to send all the neural network calculations to the GPU (and get the results back), we can't utilize it to speed up the calculations.
CUDA has been developed by NVIDIA, and it only works with NVIDIA's hardware.

However, things are happening in the field, and there are attempts at breaking the monopoly. Look up OpenCL or AMD ROCm to learn more ☺

There are GPU’s and there are GPU’s

One thing we have not addressed is the memory of the GPU:

For the GPU to process data, it has to be placed in the dedicated GPU memory. This means that both the parameters of the model, as well as the data in the given batch, have to be moved to the GPU memory prior to processing.

Moving the information takes time, so we do not want to, for instance, move part of the model back and forth during backpropagation. So, we generally can not train a model using the GPU, if the entire model (including graph) can not fit in its memory.

Let’s compare different GPU’s on memory:

Laptops:

MX250: 2 GB RTX 3050: 4 GB RTX 3070: 8 GB

Google Colab:

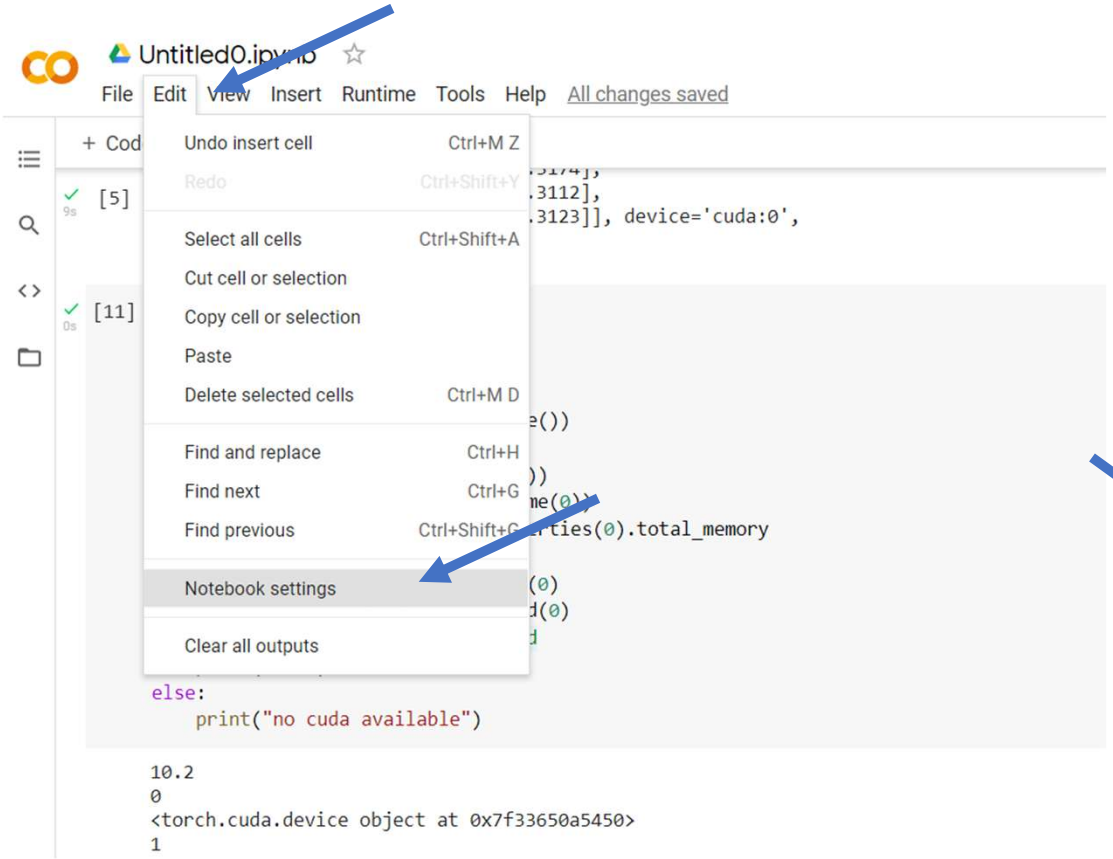
Tesla K80: 12 GB

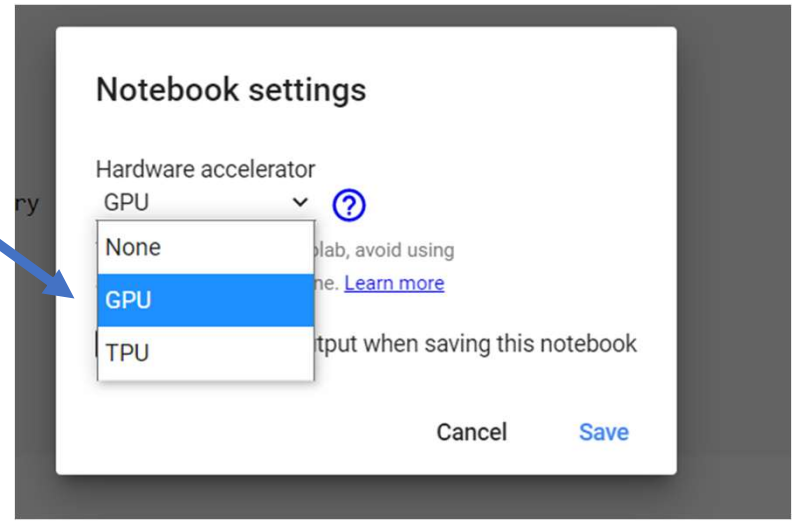
Datacenter GPUs:

NVIDIA A100: 40 GB AMD MI250x: 2 x 64 GB <- LUMI

Depending on the model, the memory requirements can vary a lot.

Activating the GPU in google colab:





Pytorch basics

Tensors

All pytorch operations should be performed on *tensors*. They are a general-purpose data container like the numpy array, and accepts many of the same operations.

The primary difference between tensors and all other containers is that they keep track of gradients, and can be moved to the GPU.

Note:

'Tensors' are, generally speaking, an advanced mathematical construct, and the precise meaning may differ somewhat between fields. However, in deep learning it simply means 'multidimensional matrix'.

```
import torch
import numpy as np


d1=np.random.rand(2,3)
print('np array:')
print(d1)

t1=torch.tensor(d1)
print('tensor:')
print(t1)

d2=t1.numpy()
print('back to np:')
print(d2)
```

```
np array:
[[0.9164676  0.85519858 0.49744912]
 [0.06946489 0.33711531 0.51941202]]
tensor:
tensor([[0.9165, 0.8552, 0.4974],
        [0.0695, 0.3371, 0.5194]], dtype=torch.float64)
back to np:
[[0.9164676  0.85519858 0.49744912]
 [0.06946489 0.33711531 0.51941202]]
```

Tensor basics

Tensors are simple to create:

```
tensorOfZeros=torch.zeros([2, 4], dtype=torch.int32)
print(tensorOfZeros)
```

```
tensor([[0, 0, 0, 0],
        [0, 0, 0, 0]], dtype=torch.int32)
```

```
tensorOfnumbers=torch.tensor([1,2,10], dtype=torch.int32)
print(tensorOfnumbers)
```

```
tensor([ 1,  2, 10], dtype=torch.int32)
```

```
npArray=np.random.rand(2,3)
tensorFromNumpy=torch.tensor(npArray)
print(tensorFromNumpy)
```

```
tensor([[0.3579, 0.3389, 0.5618],
        [0.9189, 0.0051, 0.4135]], dtype=torch.float64)
```

And we can perform the same operations as we are used to from numpy:

```
newShape=tensorFromNumpy.reshape((3,2))
print(newShape)
```

```
tensor([[0.3579, 0.3389],
        [0.5618, 0.9189],
        [0.0051, 0.4135]], dtype=torch.float64)
```

```
print(newShape.sum(dim=0))
```

```
tensor([0.9248, 1.6713], dtype=torch.float64)
```

```
print(newShape.float())
```

```
tensor([[0.3579, 0.3389],
        [0.5618, 0.9189],
        [0.0051, 0.4135]])
```

Further reading:
https://pytorch.org/docs/stable/tensors.html
https://pytorch.org/docs/stable/torch.html#indexing-slicing-joining

Tensor basics - autograd

The reason why pytorch and tensors are actually smart, is autograd:

```
w=torch.tensor(2.0, requires_grad=True)
x=torch.rand((2,2))
x2=x*w
L=torch.sum(x2,axis=(0,1))
```

We're interested in the gradient of L with respect to w.
This will be saved in w.grad:

```
print(w.grad)
```

```
None
```

But it's not there yet?

We have to call 'backward' on L,
to calculate dL/dw:

```
L.retain_grad()
x2.retain_grad()
L.backward(retain_graph=True
print(w.grad)
```

```
tensor(2.3276)
```

Because we called 'retain_grad()', we can see what else has gradients:

```
print(x.grad)
print(x2.grad)
print(L.grad)
```

```
None
tensor([[1., 1.],
        [1., 1.]])
tensor(1.)
```

All we have to do is tell pytorch whether a tensor should calculate it's gradient.

When we wish to backpropagate, we call .backward() on the last (scalar) output, and gradients are computed for all componenets leading up to it.

.retain_grad() is used here to make it possible to inspect x2.grad and L.grad, which would otherwise be None.

Dataset and dataloader

Another major component is the data handling – deep learning is nothing without appropriate input data.

Pytorch contains two very useful abstractions: **Dataset** and **DataLoader**.

Dataset is an abstract class with inbuilt functions for handling large datasets (that need to be loaded piecewise) and has several prebuilt tools for easy handling (such as TensorDataset for automatic tensor-> Dataset creation).

*DataLoader*s iterate over Datasets. They handle batching and may use multiple cpus to quickly load the data to feed it to the networks.

```
#tensor with 2000 data points, each data point is a 3x4 random matrix:
data_tensor_X=torch.rand(2000,3,4)

#ylabels (targets), 2000 integers from 0 to 3:
data_tensor_y=torch.randint(low=0, high=4,(2000,))

#we combine X and y in the same dataset:
trainDataset=torch.utils.data.TensorDataset(data_tensor_X,data_tensor_y)

trainDataLoader=torch.utils.dataLoader(trainDataset,batch_size=32)

#it is now very easy to iterate over the training data:
for xbatch,ybatch in trainDataLoader:
  y_pred=net.forward(xbatch) #calculate network output
  loss=loss_fn(y_pred,ybatch) #calculate loss for this output

  net.zero_grad()
  loss.backward() #perform backpropagation (calculate gradients)
  optimizer.step() #update weights according to the gradients

#(the above would just be a single training epoch)
```

Datasets

The simplest way to create a dataset:

torch.utils.data.TensorDataset(*tensors)

However, we can also write our own general thing:

```
class myVeryOwnDataSetClass(torch.utils.data.Dataset):
    def __init__(self, initOptions):
        #uses initOptions to initialize our dataset

    def __len__(self):
        #some way of counting elements in the data set
        return "number of elements we could possibly load"

    def __getitem__(self, idx):
        #this is where the magic happens (usually).
        # here we load files, pick the important bits,
        # transform, filter etc.

        return "data point requested"
```

Read more: https://pytorch.org/docs/stable/data.html

Special mention goes to
torchvision.datasets.DatasetFolder
torchvision.datasets.ImageFolder

Given a folder structure with all examples of a given class in separate folders, these will create data sets and loaders for us.
Additionally, torchvision has a long list of pre-made image datasets, ready to be loaded into pytorch.
See https://pytorch.org/vision/stable/datasets.html.

Once we have a dataset, we can do a bunch of things: random_split, concatenation, subset etc.

Dataloaders

When we have a dataset, we still need to actually load the the data points (if it's not all preloaded), and sample it in minibatches. We do this with **dataloaders**:

torch.utils.data.DataLoader(dataset, batch_size=1, shuffle=False, num_workers=0, drop_last=False)

Returns an iterator over the dataset

Batch_size: how many datapoints to load at once

Shuffle: randomize the order of data points between each epoch (generally a good idea!)

Num_workers: how many extra cpu's to task with loading data (data load can be time consuming)

Drop_last: if len(dataset) is not a multiple of batch_size, we either return a smaller batch, or drop the overflow

See the full syntax at: https://pytorch.org/docs/stable/data.html

(note that num_workers>0 is a lot more reliable on linux machines than windows...)

nn.Module – the actual network!

This is the base class for neural networks, as implemented in Pytorch. All your networks will be made as subclasses of nn.Module.

All common network ingredients can be imported from the torch.nn library (more on that later).

```
import torch
import torch.nn as nn

class NeuralNetwork(nn.Module):
  def __init__(self):
    super().__init__() #initialize parent class

    #input our own initializer (define the network layers)

    def forward(self,x):
      #define what the network should do with x to turn it into output
      return output
```

When we have implemented everything in pytorch, backpropagation is as simple as calling .backward() on the loss function.

```
net=NeuralNetwork() #create a network
optimizer=torch.optim.SGD(net.params(), lr=0.001)  #setup optimizer
loss_fn = nn.CrossEntropyLoss()

for iEpoch in range(nEpochs):
  for x_minibatch,y_minibatch in dataLoader:
    y_pred=net.forward(x_miniBatch) #calculate network output
    loss=loss_fn(y_pred,y_miniBatch) #calculate loss for this output

    net.zero_grad()
    loss.backward() #perform backpropagation (calculate gradients)
    optimizer.step() #update weights according to the gradients
```

nn.Sequential

In the simple case where our network is simply a bunch of layers stacked on top of each other, pytorch gives us a simpler syntax:

```
import torch
from torch import nn

model = nn.Sequential(
          nn.Linear(20,30), #input 20, output 30
          nn.ReLU(), #activation function
          nn.Linear(30,4), #input 30, output 4
          nn.Softmax(dim=1) #softmax across columns
        )

randomInput=torch.rand((3,20))
print(model(randomInput))
```

```
tensor([[0.3144, 0.2526, 0.2146, 0.2184],
        [0.2672, 0.2457, 0.2293, 0.2578],
        [0.2735, 0.2286, 0.2221, 0.2758]], grad_fn=<SoftmaxBackward>)
```

Note how nn.Sequential does not output a class, but an actual network, ready for inputs.

While nn.Sequential is handy, I recommend getting familiar with the general syntax. *Most advanced networks are not sequential.*

Read more: https://pytorch.org/docs/stable/generated/torch.nn.Sequential.html

.to(device)

Until now, we haven't actually used the GPU. This is done by moving elements to the GPU memory:

```
device = torch.device('cuda')

model.to(device)
randomInputOnGPU=randomInput.to(device)

model(randomInputOnGPU) #will be performed with GPU
```

```
tensor([[0.3045, 0.2511, 0.2274, 0.2171],
        [0.3059, 0.2491, 0.2320, 0.2129],
        [0.2879, 0.2665, 0.2377, 0.2080]], device='cuda:0',
       grad_fn=<SoftmaxBackward>)
```

Note that if we overwrite 'randomInputOnGPU', that GPU memory is freed.

Read more: https://pytorch.org/docs/stable/notes/cuda.html

A very useful bit of code to have in the beginning of your script is:

```
device=torch.device('cpu')
if torch.cuda.is_available():
    cuda=torch.device('cuda:0')
    print(torch.version.cuda)
    print(torch.cuda.current_device())
    print(torch.cuda.device(0))
    print(torch.cuda.device_count())
    print(torch.cuda.get_device_name(0))
    t = torch.cuda.get_device_properties(0).total_memory
    print(t)
    r = torch.cuda.memory_reserved(0)
    a = torch.cuda.memory_allocated(0)
    f = r-a  # free inside reserved
    print(f/1e6)
else:
    print("no cuda available")
```

```
10.2
0
<torch.cuda.device object at 0x7f33650a5450>
1
Tesla K80
11996954624
2.090496
```

Training

```
model=myModelClass()
model.to(device) #moving model to GPU, prior to optimizer setup
optimizer = torch.optim.SGD(model.parameters(), lr=learningRate)
loss_fn=nn.functional.cross_entropy

model.train()
for iEpoch in range(nEpoch):
  for xbatch,ybatch in trainDataLoader:
    xbatch=xbatch.to(device) #moving input to GPU
    ybatch=ybatch.to(device) #moving targets to GPU
    y_pred=model(xbatch)

    loss = loss_fn(y_pred,ybatch)

    # Zero the gradients before running the backward pass.
    model.zero_grad()
    #repopulate gradients
    loss.backward()
    #update weights
    optimizer.step()
```

Note how we are passing the 'model.parameters()' to the optimizer. This tells it which tensors's gradients to inspect later.

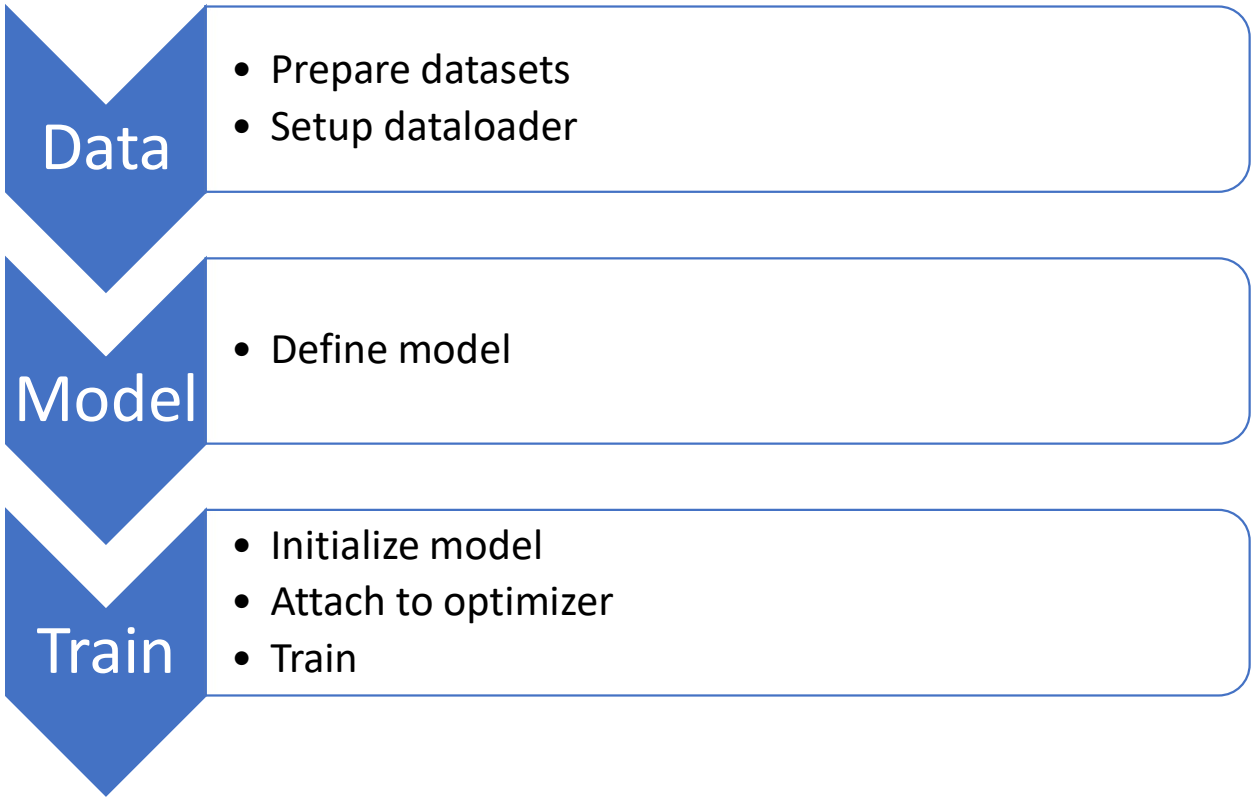
You don't need to use the GPU this week, but we are showing you where the good places to move data are (in your code).

Note that you have to move the output back to the cpu if you want to do further analysis with it (send it to numpy for instance):

```
y_pred.detach().to('cpu').numpy()
```

Anatomy of a pytorch script

The ‘data’ and ‘model’ steps can be interchanged depending on preference.



Coffee / tea break?

Then: this week’s exercise

Logistic regression in pytorch

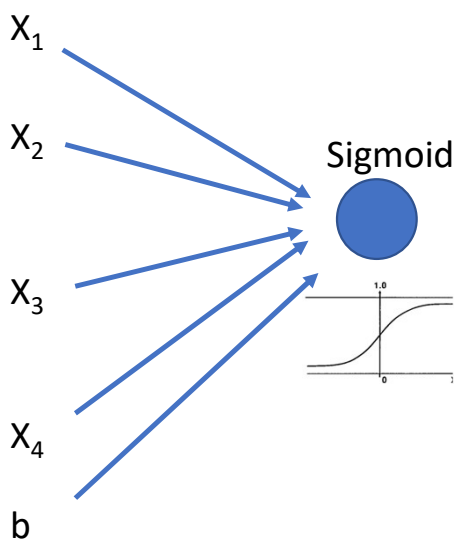
Let us walk through last week's exercise, but in pytorch:

We use the same MNIST data as before, but this time we package it in tensors:

```
from mlxtend.data import mnist_data
X, y = mnist_data()

#we just focus on 1's and 0's:
keep=(y==0) | (y==1)
X=X[keep,:]
y=y[keep]

#create test and training data:
Xtrain,Xtest,ytrain,ytest=sklearn.model_selection.train_test_split(X,y,test_size=0.3,shuffle=True)

#transform into pytorch datasets:
trainDataset=torch.utils.data.TensorDataset(torch.tensor(Xtrain).float(),torch.tensor(ytrain).float())
testDataset=torch.utils.data.TensorDataset(torch.tensor(Xtest).float(),torch.tensor(ytest).float())
```

Let’s make a network:

We remember that logistic regression is a linear combination of all inputs, fed to a sigmoid:




```
class logRegression(nn.Module):
    def __init__(self):
        super().__init__()
        self.inputLayer=nn.Linear(784, 1, bias=True)

    def forward(self,x):
      x=self.inputLayer(x)
      x=torch.sigmoid(x)

      return torch.squeeze(x)
```

Note how we get a standard layer from ’nn’, and use the inbuilt sigmoid from torch.

We can make a quick test that there are no obvious problems:

```
testData=torch.rand((20,784))
testNet=logRegression()
testNet.forward(testData)
```

```
tensor([0.5577, 0.6076, 0.5991, 0.6446, 0.6048, 0.5628, 0.5934, 0.5725, 0.6369,
        0.6492, 0.6280, 0.5308, 0.6694, 0.6368, 0.5486, 0.6008, 0.6377, 0.5888,
        0.6007, 0.5607], grad_fn=<SqueezeBackward0>)
```

Setup a network for training, including loss function and the SGD optimizer:

```
#setup training and run
nEpoch=10
learningRate=0.001
batchSize=64

trainDataLoader=torch.utils.data.DataLoader(trainDataset,batch_size=batchSize,shuffle=True)

model=logRegression()

optimizer = torch.optim.SGD(model.parameters(), lr=learningRate)

#we use the binary cross entropy loss (like we did in the numpy version)
loss_fn=nn.BCELoss()
```

Do the actual training:

```
accuracy=np.zeros(nEpoch)
for iEpoch in range(nEpoch):
  estimatedLabels=np.empty((0,))
  trueLabels=np.empty((0,))
  for xbatch,ybatch in trainDataLoader:
    y_pred=model(xbatch)

    loss = loss_fn(y_pred,ybatch)

    # Zero the gradients before running the backward pass.
    model.zero_grad()

    loss.backward()

    optimizer.step()


    #calculate accuracy for whole training set:
    yhat=y_pred>.5
    estimatedLabels=np.append(estimatedLabels,yhat)
    trueLabels=np.append(trueLabels,ybatch)
  accuracy[iEpoch]=np.mean(estimatedLabels==trueLabels)
```

When we print ‘accuracy’, and evaluate on the test data, it’s really very good:

```
accuracy
```

```
array([0.93285714, 0.99428571, 0.99857143, 0.99857143, 0.99857143,
       0.99857143, 0.99857143, 0.99857143, 0.99857143, 0.99857143])
```

```
#test performance:
testDataLoader=torch.utils.data.DataLoader(testDataset,batch_size=batchSize,shuffle=True)

estimatedLabels=np.empty((0,))
trueLabels=np.empty((0,))
for xbatch,ybatch in testDataLoader:
    y_pred=model(xbatch)


    #calculate accuracy for whole training set:
    yhat=y_pred>.5
    estimatedLabels=np.append(estimatedLabels,yhat)
    trueLabels=np.append(trueLabels,ybatch)

testAcc=np.mean(estimatedLabels==trueLabels)
print(testAcc)
```

```
0.9866666666666667
```

This week’s reading and exercises

Reading

Required reading:

https://pytorch.org/tutorials/beginner/basics/data_tutorial.html

Suggested reading:

- Video on computational graphs: https://www.youtube.com/watch?v=d14TUNcbn1k
- Pytorch autograd: https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html
- More on torch.utils.data: https://pytorch.org/docs/stable/data.html
- Official tutorials: https://pytorch.org/tutorials/
- Note on Brightspace: ‘proof’ regarding posterior probabilities

Exercise: Perform digit recognition for all 10 types of digit in MNIST

A: Reproduce the code for two digits as shown in the slides

B: Try extending the code to work for 10 classes: use all data points, change the sigmoid to a softmax, and look at nn.functional.cross_entropy.

C: Most likely, that has poor performance. Try adding more layers to the network, and optionally change the ‘optimizer’ to ‘adam’ instead of 'SGD'. A high accuracy (>95%) is possible, using only the methods covered thus far.