Artificial Intelligence and Machine Learning

Unit II

Multi-Layer Perceptron and Introduction to Deep Learning

My own latex definitions

In [1]:
```
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline
plt.style.use('seaborn-whitegrid')

font = {'family' : 'Times',
        'weight' : 'bold',
        'size'   : 12}

matplotlib.rc('font', **font)


# Aux functions

def plot_grid(Xs, Ys, axs=None):
    ''' Aux function to plot a grid'''
    t = np.arange(Xs.size) # define progression of int for indexing colormap
    if axs:
        axs.plot(0, 0, marker='*', color='r', linestyle='none') #plot origin
        axs.scatter(Xs,Ys, c=t, cmap='jet', marker='.') # scatter x vs y
        axs.axis('scaled') # axis scaled
    else:
        plt.plot(0, 0, marker='*', color='r', linestyle='none') #plot origin
        plt.scatter(Xs,Ys, c=t, cmap='jet', marker='.') # scatter x vs y
        plt.axis('scaled') # axis scaled

def linear_map(A, Xs, Ys):
    '''Map src points with A'''
    # [NxN,NxN] -> NxNx2 # add 3-rd axis, like adding another layer
    src = np.stack((Xs,Ys), axis=Xs.ndim)
    # flatten first two dimension
    # (NN)x2
    src_r = src.reshape(-1,src.shape[-1]) #ask reshape to keep last dimension and adjust the rest
    # 2x2 @ 2x(NN)
    dst = A @ src_r.T # 2xNN
    #(NN)x2 and then reshape as NxNx2
    dst = (dst.T).reshape(src.shape)
    # Access X and Y
    return dst[...,0], dst[...,1]


def plot_points(ax, Xs, Ys, col='red', unit=None, linestyle='solid'):
    '''Plots points'''
    ax.set_aspect('equal')
    ax.grid(True, which='both')
    ax.axhline(y=0, color='gray', linestyle="--")
    ax.axvline(x=0, color='gray',  linestyle="--")
    ax.plot(Xs, Ys, color=col)
    if unit is None:
        plotVectors(ax, [[0,1],[1,0]], ['gray']*2, alpha=1, linestyle=linestyle)
    else:
        plotVectors(ax, unit, [col]*2, alpha=1, linestyle=linestyle)

def plotVectors(ax, vecs, cols, alpha=1, linestyle='solid'):
    '''Plot set of vectors.'''
    for i in range(len(vecs)):
        x = np.concatenate([[0,0], vecs[i]])
        ax.quiver([x[0]],
                  [x[1]],
                  [x[2]],
                  [x[3]],
                  angles='xy', scale_units='xy', scale=1, color=cols[i],
                  alpha=alpha, linestyle=linestyle, linewidth=2)
```

```
/var/folders/rt/lg7n4lt1489270pz_18qn1_c0000gp/T/ipykernel_61746/1496334134.py:5: MatplotlibDeprecationWarning:
The seaborn styles shipped by Matplotlib are deprecated since 3.6, as they no longer correspond to the styles s
hipped by seaborn. However, they will remain available as 'seaborn-v0_8-<style>'. Alternatively, directly use t
he seaborn API instead.
  plt.style.use('seaborn-whitegrid')
```

In [2]:
```
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline
plt.style.use('seaborn-whitegrid')

font = {'family' : 'Times',
        'weight' : 'bold',
        'size'   : 12}

matplotlib.rc('font', **font)


# Aux functions

def plot_grid(Xs, Ys, axs=None):
    ''' Aux function to plot a grid'''
    t = np.arange(Xs.size) # define progression of int for indexing colormap
    if axs:
        axs.plot(0, 0, marker='*', color='r', linestyle='none') #plot origin
        axs.scatter(Xs,Ys, c=t, cmap='jet', marker='.') # scatter x vs y
        axs.axis('scaled') # axis scaled
    else:
        plt.plot(0, 0, marker='*', color='r', linestyle='none') #plot origin
        plt.scatter(Xs,Ys, c=t, cmap='jet', marker='.') # scatter x vs y
        plt.axis('scaled') # axis scaled

def linear_map(A, Xs, Ys):
    '''Map src points with A'''
    # [NxN,NxN] -> NxNx2 # add 3-rd axis, like adding another layer
    src = np.stack((Xs,Ys), axis=Xs.ndim)
    # flatten first two dimension
    # (NN)x2
    src_r = src.reshape(-1,src.shape[-1]) #ask reshape to keep last dimension and adjust the rest
    # 2x2 @ 2x(NN)
    dst = A @ src_r.T # 2xNN
    #(NN)x2 and then reshape as NxNx2
    dst = (dst.T).reshape(src.shape)
    # Access X and Y
    return dst[...,0], dst[...,1]


def plot_points(ax, Xs, Ys, col='red', unit=None, linestyle='solid'):
    '''Plots points'''
    ax.set_aspect('equal')
    ax.grid(True, which='both')
    ax.axhline(y=0, color='gray', linestyle="--")
    ax.axvline(x=0, color='gray',  linestyle="--")
    ax.plot(Xs, Ys, color=col)
    if unit is None:
        plotVectors(ax, [[0,1],[1,0]], ['gray']*2, alpha=1, linestyle=linestyle)
    else:
        plotVectors(ax, unit, [col]*2, alpha=1, linestyle=linestyle)

def plotVectors(ax, vecs, cols, alpha=1, linestyle='solid'):
    '''Plot set of vectors.'''
    for i in range(len(vecs)):
        x = np.concatenate([[0,0], vecs[i]])
        ax.quiver([x[0]],
                  [x[1]],
                  [x[2]],
                  [x[3]],
                  angles='xy', scale_units='xy', scale=1, color=cols[i],
                  alpha=alpha, linestyle=linestyle, linewidth=2)
```

```
/var/folders/rt/lg7n4lt1489270pz_18qn1_c0000gp/T/ipykernel_61746/1496334134.py:5: MatplotlibDeprecationWarning:
The seaborn styles shipped by Matplotlib are deprecated since 3.6, as they no longer correspond to the styles s
hipped by seaborn. However, they will remain available as 'seaborn-v0_8-<style>'. Alternatively, directly use t
he seaborn API instead.
  plt.style.use('seaborn-whitegrid')
```

Recap previous lecture

- Multi-Class Classification
- **SoftMax Regression** plus **Cross-Entropy Loss**

Today's lecture

Supervised, Parametric Models

Propaedeutic part for Deep Learning

0) Optimization in Deep Learning

1) Network Structure: Multi-Layer Perceptron (MLP) is a Fully-Connected Neural Net

2) Backpropagation

This lecture material is taken from

- d2l.ai - Multi Variable Calculus
- Karpathy (Tesla Machine Learning Directory) Lecture on Backprop
- Stanford Neural Nets and Backprop lecture
- Stanford ML notes on Neural Nets
- Stanford ML notes on Backprop
- Animation from jermwatt.github.io

Deep Learning

0) Quick Intro to Optimization in Deep Learning

1) What is a Neural Net (just Multi-Layer Perceptron)

2) How to obtain gradients on the weights

Gradient Descent or Batch GD

- Compute the gradient of the loss wrt params for **all n training samples**
- $\boldsymbol{\theta} - \gamma \frac{1}{n} \sum_{i=1}^{n} \nabla_{\boldsymbol{\theta}} \mathrm{J}(\theta; \mathbf{x}_i, y_i)$

Stochastic Gradient Descent or SGD

- Compute the gradient of the loss wrt params for **a single random training sample**
- $\boldsymbol{\theta} - \gamma \nabla_{\boldsymbol{\theta}} \mathrm{J}(\theta; \mathbf{x}_i, y_i)$

How to optimize a Neural Net - SGD over mini-batches

1. In-between Batch GD and SGD with a single sample
2. We load randomly k samples over the n; usually k is a power of 2.
   - mini batch of `32, 64, 128` but could also be `100`
3. $\boldsymbol{\theta} - \gamma \frac{1}{k} \sum_{i=1}^{k} \nabla_{\boldsymbol{\theta}} \mathrm{J}(\theta; \mathbf{x}_i, y_i)$
4. Practically you take your training set X and you **shuffle** it, then go over it k by k. *Simulate uniform random sampling without replacement.*
   - When the list is over, re-start and shuffle again.
5. When you have perfomed a full pass on the shuffled data, this is called an **EPOCH**
6. You can train NN over iterations or over **EPOCHS**

NN training scheme - Pseudo-code

```
from random import shuffle
training = list(range(1,11)) # each index points to a training sample, could be a matrix x=HxWx3,
label y
shuffle(training)
converge, it, max_it, k, epoch = False, 0, 100, 3, 0
while not converge and it < max_it: # you training convergence scheme
    print(f'[Epoch {epoch}]')
    for b in range(0, len(training), k): # Data Loader gives you a batch k x matrices
        mini_batch = training[b:b+k] # so mini-batch is a tensor HxWx3xk
        if len(mini_batch) != k: # a possible way of handling the offset
            continue
        print('SGD step taken over', mini_batch) # compute the loss/gradients and upate your model
        loss.backward()  # get the gradients
        optimizer.step() # incorporate in the model
        # check convergence and set it to True
        it += 1
    epoch += 1 # an epoch is done, we reshuffle the training set
    shuffle(training)

> Original unshuffled training set [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
> Training set [10, 8, 1, 2, 6, 4, 9, 7, 5, 3]
[Epoch 0]
SGD step taken over [10, 8, 1]
SGD step taken over [2, 6, 4]
SGD step taken over [9, 7, 5]
> Training set [1, 10, 6, 9, 3, 7, 8, 4, 5, 2]
[Epoch 1]
SGD step taken over [1, 10, 6]
SGD step taken over [9, 3, 7]
SGD step taken over [8, 4, 5]
> Training set [6, 3, 10, 5, 9, 8, 4, 7, 2, 1]
[Epoch 2]
SGD step taken over [6, 3, 10]
SGD step taken over [5, 9, 8]
SGD step taken over [4, 7, 2]
> Training set [1, 2, 5, 10, 6, 7, 9, 8, 3, 4]
[Epoch 3]
SGD step taken over [1, 2, 5]
SGD step taken over [10, 6, 7]
SGD step taken over [9, 8, 3]
> Training set [2, 3, 1, 9, 6, 8, 4, 10, 7, 5]
[Epoch 4]
```

Images - Mini-batch is a tensor `HxWx3xk`

as an example with RGB images of size $H \times W$, you have a tensor that contains k images in the mini-batch

Video Frames - Mini-batch is a tensor `HxWx3xtxk`

as set of frames from a video, you have a tensor that contains k frames over t time instants of the videos in the mini-batch

In [3]:
```
from random import shuffle
training = list(range(1,11))
shuffle(training)
print('> Training set', training)
converge, it, max_it, k, epoch = False, 0, 100, 3, 0
while not converge and it < max_it: # you training converge scheme
    print(f'[Epoch {epoch}]')
    for b in range(0, len(training), k): # Data Loader
        mini_batch = training[b:b+k]
        if len(mini_batch) != k:
            continue
        print('SGD step taken over', mini_batch) # compute the loss and upate your model
        it += 1
    epoch += 1 # epoch is done we reshuffle the training set
    shuffle(training)
    print('> Training set', training)
```

```
> Training set [3, 7, 8, 2, 10, 5, 4, 6, 1, 9]
[Epoch 0]
SGD step taken over [3, 7, 8]
SGD step taken over [2, 10, 5]
SGD step taken over [4, 6, 1]
> Training set [1, 7, 5, 2, 4, 6, 8, 10, 3, 9]
[Epoch 1]
SGD step taken over [1, 7, 5]
SGD step taken over [2, 4, 6]
SGD step taken over [8, 10, 3]
> Training set [3, 8, 6, 1, 9, 2, 7, 4, 10, 5]
[Epoch 2]
SGD step taken over [3, 8, 6]
SGD step taken over [1, 9, 2]
SGD step taken over [7, 4, 10]
> Training set [3, 10, 2, 6, 5, 8, 9, 1, 4, 7]
[Epoch 3]
SGD step taken over [3, 10, 2]
SGD step taken over [6, 5, 8]
SGD step taken over [9, 1, 4]
> Training set [3, 2, 1, 6, 4, 5, 8, 9, 10, 7]
[Epoch 4]
SGD step taken over [3, 2, 1]
SGD step taken over [6, 4, 5]
SGD step taken over [8, 9, 10]
> Training set [1, 10, 7, 5, 3, 2, 8, 9, 4, 6]
[Epoch 5]
SGD step taken over [1, 10, 7]
SGD step taken over [5, 3, 2]
SGD step taken over [8, 9, 4]
> Training set [10, 3, 8, 6, 2, 5, 9, 7, 4, 1]
[Epoch 6]
SGD step taken over [10, 3, 8]
SGD step taken over [6, 2, 5]
SGD step taken over [9, 7, 4]
> Training set [1, 6, 9, 7, 5, 3, 10, 4, 2, 8]
[Epoch 7]
SGD step taken over [1, 6, 9]
SGD step taken over [7, 5, 3]
SGD step taken over [10, 4, 2]
> Training set [4, 7, 3, 8, 9, 5, 2, 6, 10, 1]
[Epoch 8]
SGD step taken over [4, 7, 3]
SGD step taken over [8, 9, 5]
SGD step taken over [2, 6, 10]
> Training set [10, 1, 3, 9, 8, 5, 6, 2, 7, 4]
[Epoch 9]
SGD step taken over [10, 1, 3]
SGD step taken over [9, 8, 5]
SGD step taken over [6, 2, 7]
> Training set [5, 6, 10, 7, 4, 3, 8, 9, 1, 2]
[Epoch 10]
SGD step taken over [5, 6, 10]
SGD step taken over [7, 4, 3]
SGD step taken over [8, 9, 1]
> Training set [4, 3, 5, 9, 6, 2, 8, 10, 7, 1]
[Epoch 11]
SGD step taken over [4, 3, 5]
SGD step taken over [9, 6, 2]
SGD step taken over [8, 10, 7]
> Training set [4, 8, 2, 5, 10, 3, 7, 9, 1, 6]
[Epoch 12]
SGD step taken over [4, 8, 2]
SGD step taken over [5, 10, 3]
SGD step taken over [7, 9, 1]
> Training set [2, 8, 4, 1, 9, 3, 5, 6, 10, 7]
[Epoch 13]
SGD step taken over [2, 8, 4]
SGD step taken over [1, 9, 3]
SGD step taken over [5, 6, 10]
> Training set [3, 7, 10, 5, 4, 9, 1, 2, 6, 8]
[Epoch 14]
SGD step taken over [3, 7, 10]
SGD step taken over [5, 4, 9]
SGD step taken over [1, 2, 6]
> Training set [4, 2, 7, 5, 8, 10, 9, 6, 3, 1]
[Epoch 15]
SGD step taken over [4, 2, 7]
SGD step taken over [5, 8, 10]
SGD step taken over [9, 6, 3]
> Training set [3, 5, 7, 6, 8, 9, 4, 2, 1, 10]
[Epoch 16]
SGD step taken over [3, 5, 7]
SGD step taken over [6, 8, 9]
SGD step taken over [4, 2, 1]
> Training set [8, 1, 10, 2, 5, 3, 4, 7, 6, 9]
[Epoch 17]
SGD step taken over [8, 1, 10]
SGD step taken over [2, 5, 3]
SGD step taken over [4, 7, 6]
```

```
> Training set [10, 9, 6, 4, 7, 2, 1, 5, 8, 3]
[Epoch 18]
SGD step taken over [10, 9, 6]
SGD step taken over [4, 7, 2]
SGD step taken over [1, 5, 8]
> Training set [4, 10, 6, 2, 7, 9, 3, 8, 5, 1]
[Epoch 19]
SGD step taken over [4, 10, 6]
SGD step taken over [2, 7, 9]
SGD step taken over [3, 8, 5]
> Training set [5, 8, 2, 10, 4, 9, 7, 1, 6, 3]
[Epoch 20]
SGD step taken over [5, 8, 2]
SGD step taken over [10, 4, 9]
SGD step taken over [7, 1, 6]
> Training set [3, 6, 10, 4, 9, 8, 1, 7, 5, 2]
[Epoch 21]
SGD step taken over [3, 6, 10]
SGD step taken over [4, 9, 8]
SGD step taken over [1, 7, 5]
> Training set [7, 9, 5, 6, 3, 4, 10, 8, 2, 1]
[Epoch 22]
SGD step taken over [7, 9, 5]
SGD step taken over [6, 3, 4]
SGD step taken over [10, 8, 2]
> Training set [8, 2, 3, 7, 1, 4, 5, 6, 10, 9]
[Epoch 23]
SGD step taken over [8, 2, 3]
SGD step taken over [7, 1, 4]
SGD step taken over [5, 6, 10]
> Training set [2, 7, 9, 10, 5, 3, 6, 1, 8, 4]
[Epoch 24]
SGD step taken over [2, 7, 9]
SGD step taken over [10, 5, 3]
SGD step taken over [6, 1, 8]
> Training set [10, 3, 5, 7, 2, 1, 9, 8, 4, 6]
[Epoch 25]
SGD step taken over [10, 3, 5]
SGD step taken over [7, 2, 1]
SGD step taken over [9, 8, 4]
> Training set [1, 9, 2, 6, 3, 8, 4, 10, 5, 7]
[Epoch 26]
SGD step taken over [1, 9, 2]
SGD step taken over [6, 3, 8]
SGD step taken over [4, 10, 5]
> Training set [3, 7, 2, 9, 10, 1, 5, 4, 8, 6]
[Epoch 27]
SGD step taken over [3, 7, 2]
SGD step taken over [9, 10, 1]
SGD step taken over [5, 4, 8]
> Training set [1, 2, 4, 6, 9, 8, 7, 10, 5, 3]
[Epoch 28]
SGD step taken over [1, 2, 4]
SGD step taken over [6, 9, 8]
SGD step taken over [7, 10, 5]
> Training set [1, 5, 4, 9, 7, 10, 6, 2, 8, 3]
[Epoch 29]
SGD step taken over [1, 5, 4]
SGD step taken over [9, 7, 10]
SGD step taken over [6, 2, 8]
> Training set [3, 6, 7, 9, 2, 8, 5, 4, 10, 1]
[Epoch 30]
SGD step taken over [3, 6, 7]
SGD step taken over [9, 2, 8]
SGD step taken over [5, 4, 10]
> Training set [1, 7, 3, 10, 2, 4, 5, 8, 9, 6]
[Epoch 31]
SGD step taken over [1, 7, 3]
SGD step taken over [10, 2, 4]
SGD step taken over [5, 8, 9]
> Training set [2, 1, 4, 5, 9, 3, 8, 6, 7, 10]
[Epoch 32]
SGD step taken over [2, 1, 4]
SGD step taken over [5, 9, 3]
SGD step taken over [8, 6, 7]
> Training set [9, 6, 10, 7, 4, 1, 3, 5, 8, 2]
[Epoch 33]
SGD step taken over [9, 6, 10]
SGD step taken over [7, 4, 1]
SGD step taken over [3, 5, 8]
> Training set [1, 9, 10, 5, 6, 8, 4, 7, 3, 2]
```

1) SGD over mini-batches

1. **Initialization - Very Important if the function is not strictly convex** $\boldsymbol{\theta} \sim \mathrm{N}(\cdot)$ omit details for now$ With NN random initialization from a distribution (There are different methods). **We do not set them all to zero**
2. Repeat until **convergence**:
   - Compute the gradient of the loss wrt the parameters $\boldsymbol{\theta}$ given **the mini-batch**
   - Take a small step in the opposite direction of steepest ascent **(so steepest descent).**

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \gamma \frac{1}{k} \sum_{i=1}^{k} \nabla_{\boldsymbol{\theta}} \mathrm{J}(\theta; \mathbf{x}, y)$$

3. When convergence is reached, your final estimate is in $\boldsymbol{\theta}$

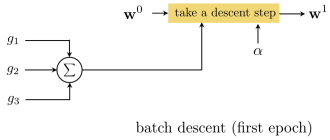
Change of vocabulary - A bunch of training samples is a mini-batch

- We train NN **Stochastic Gradient Descent** over mini-batches with momentum (or variations thereof)
- When you train NN you 'sample' a mini-batch $\mathbf{X}_b$ from your big dataset $\mathbf{X}$.

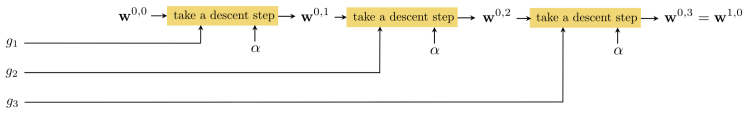
Below this holds for the final linear layer:

$$\underbrace{\mathbf{Y}}_{\mathrm{R}^{Kxn}} = \underbrace{\mathbf{W}}_{\mathrm{R}^{K\times d}} \underbrace{\mathbf{X}_b}_{\mathrm{R}^{d\times n}} + \underbrace{\mathbf{b}}_{\mathrm{R}^{K}}$$

Mini-Batch, Visually



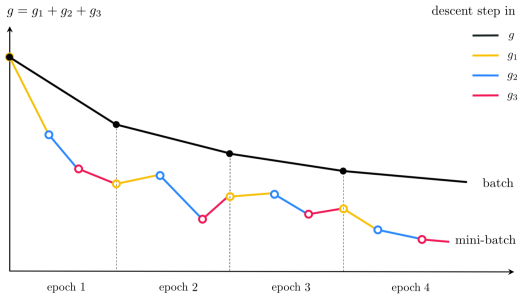
batch descent (first epoch)



stochastic descent (first epoch)

Mini-Batch SGD vs ~~[Batch]~~ GD

Loss in NN in **non-convex** with lots of local-minima so stochasticity adds noise that let the optimization escape from local minima.

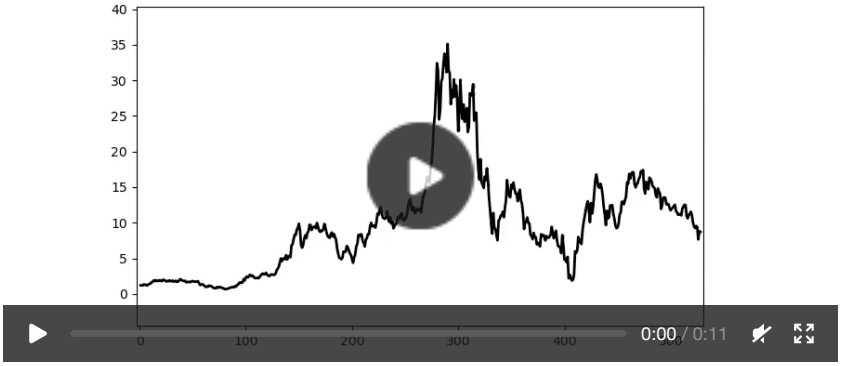


Mini-batch is a sort of smoothing of the single point SGD

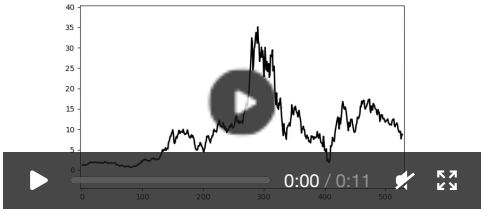
There is another smoothing technique: Momentum

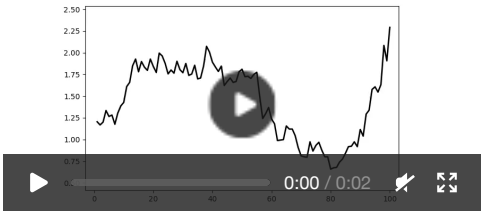
Moving average

$$\boldsymbol{\mu}_t = \frac{t-1}{t}\boldsymbol{\mu}_{t-1} + \frac{1}{t}\boldsymbol{x}_{new} \quad \text{note} \quad \frac{t-1}{t} + \frac{1}{t} = 1$$



Momentum $\approx$ exponential average





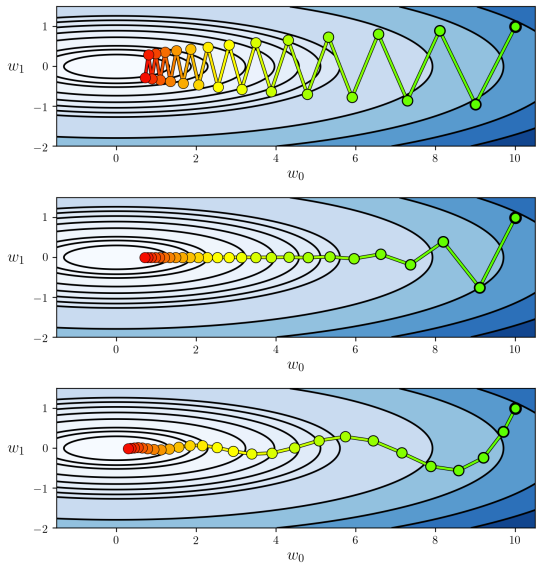
Moving average

$$\boldsymbol{\mu}_t = \frac{t-1}{t}\boldsymbol{\mu}_{t-1} + \frac{1}{t}\boldsymbol{x}_{new} \quad \text{note} \quad \frac{t-1}{t} + \frac{1}{t} = 1$$

Exponential moving average used in SGD

$$\boldsymbol{\mu}_{t+1} = \alpha\boldsymbol{\mu}_t + (1-\alpha)\boldsymbol{x}_{new} \quad 0 \le \alpha \le 1$$

Top: SGD: **Bottom:** SGD with momentum increasing memory of previous steps



SGD over mini-batches with Momentum

- We introduce an additional term **to remember what happened to the gradient in the previous iteration.**
- This memory dampens oscillations and smoothes out the gradient updates.
- The memory is implemented with a **exponential moving average**
- Usualy α (the memory param) is set to 0.9, which is a good value.

$$\Delta_t = \alpha\Delta_{t-1} + (1-\alpha)\underbrace{\frac{1}{k}\sum_{i=1}^{k}\nabla_\theta \mathrm{J}(\theta; \mathbf{x}_i, y_i)}_{\text{new update}}$$

$$\theta \leftarrow \theta - \gamma\Delta_t$$

SGD over mini-batches with Momentum

1. **Initialization - Very Important if the function is not strictly convex** $\theta \sim \mathrm{N}$ $\quad$ omit details With NN random initialization from a distribution (There are different methods). **We do not set them all to zero**
2. Repeat until **convergence**:

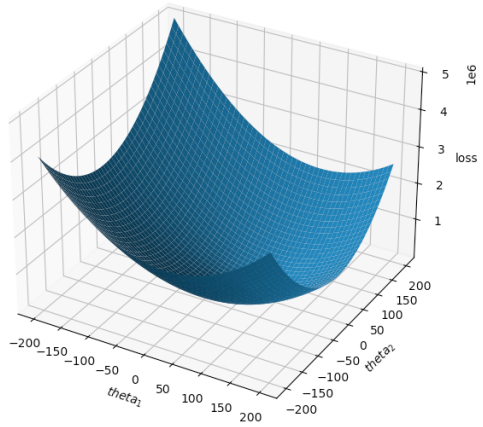
    - Compute the gradient of the loss wrt the parameters θ given **the mini-batch**
    - Take a small step in the opposite direction of steepest ascent **(so steepest descent).**

$$\Delta_{t+1} = \alpha\Delta_t + (1-\alpha)\underbrace{\frac{1}{k}\sum_{i=1}^{k}\nabla_\theta \mathrm{J}(\theta; \mathbf{x}_i, y_i)}_{\text{new update}}$$

$$\theta \leftarrow \theta - \gamma\Delta_{t+1}$$

3. When convergence is reached (or **EARLY STOPPING**), your final estimate is in θ

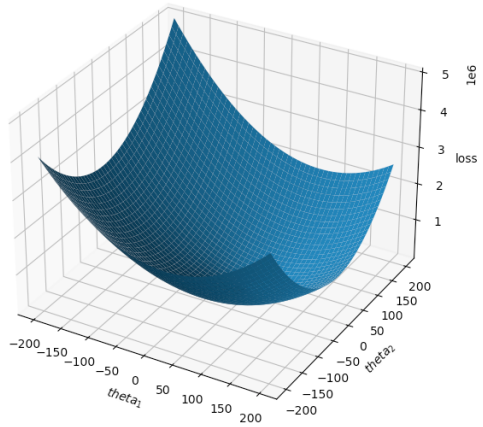
Loss Surface for Linear Regression ℓ_2^2 loss with $d = 2$ parameters in $\boldsymbol{\theta}$



Loss Surface for Linear Regression ℓ_2^2 loss with $d = 10$ parameters in $\boldsymbol{\theta}$

If you increase the dimensionality of the input data $\mathbf{x}$ and thus also those of the parameters $\boldsymbol{\theta}$ we can't draw the loss surface anymore but loss **still remains convex, which means that there is a single global optimum.**

Image the following below but in 10 dimensions!



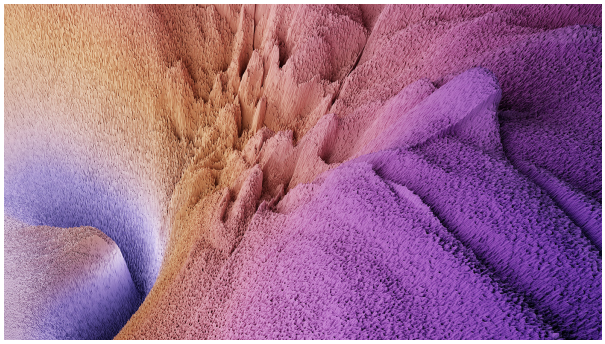
With Deep Learning optimization is highly non-convex and params explode!

Loss Surface for ResNet-20 with no skip connection on ImageNet

ResNet-20, number of parameters θ of the order of.....millions!

You will cover ResNet (residual connections) in Deep Learning course

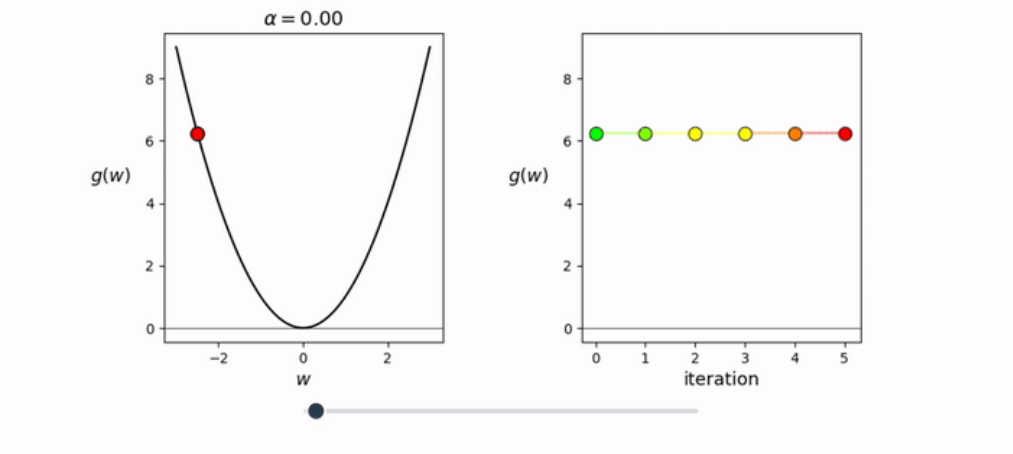
Visualization of mode connectivity for ResNet-20 with no skip connections on ImageNet dataset. The visualization by Javier Ideami



Taken from https://izmailovpavel.github.io/curves_blogpost/

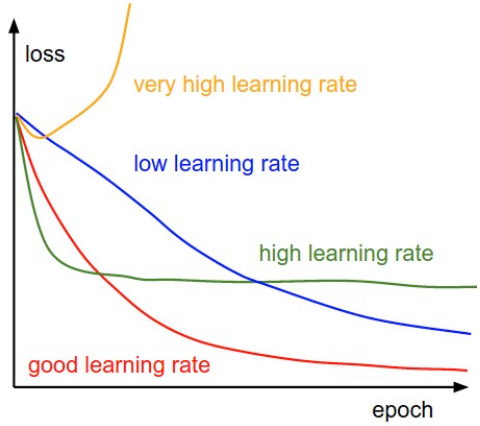
Video for the curious student

Learning rate is very important

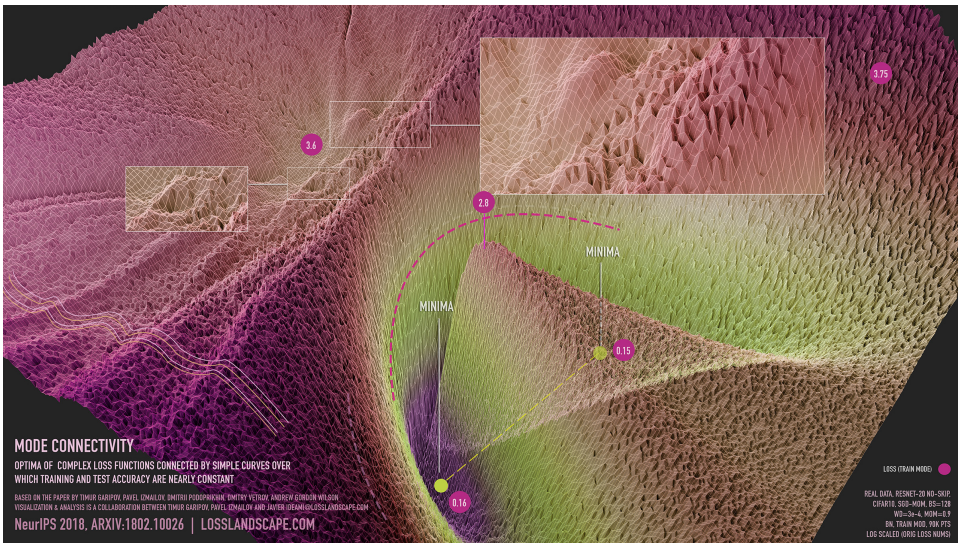


Babysitting the training process

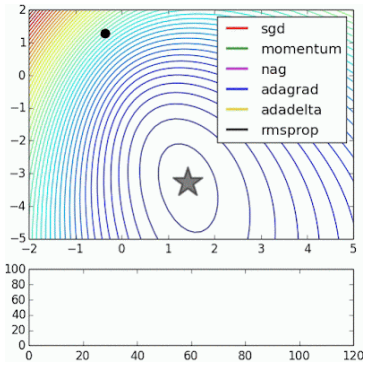
Loss in function of epochs



Valleys, Hills, Noisy Surface



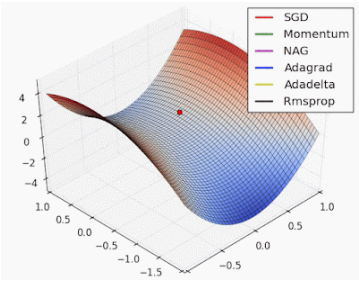
Dynamics of Training



***Noisy moons**: This is logistic regression on noisy moons dataset from sklearn which shows the smoothing effects of momentum based techniques (which also results in over shooting and correction). The error surface is visualized as an average over the whole dataset empirically, but the trajectories show the dynamics of minibatches on noisy data. The bottom chart is an accuracy plot.*

taken from here

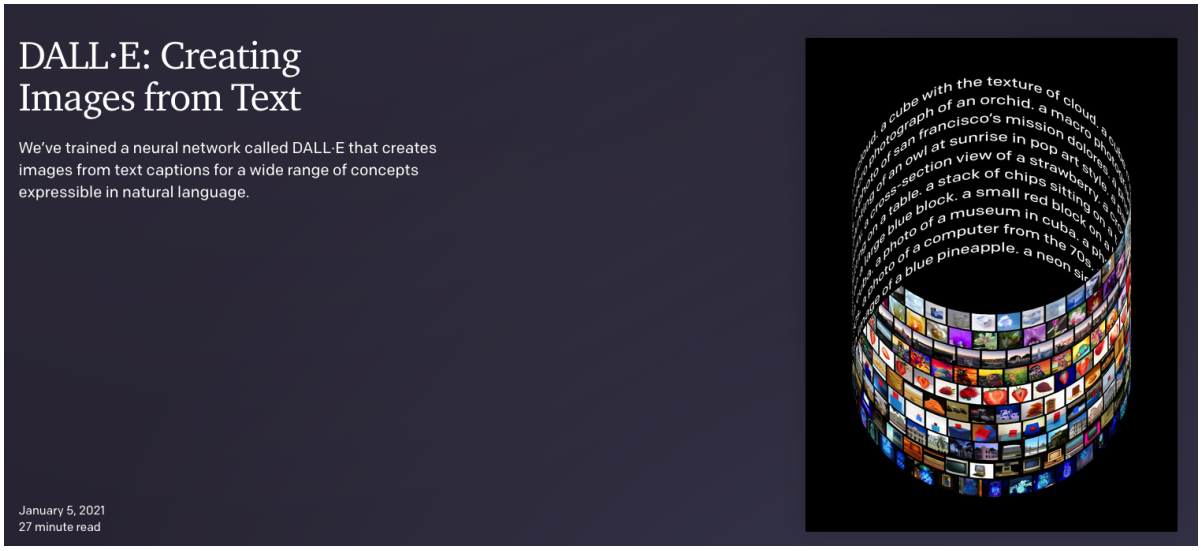
Dynamics of Training



***Long valley**: Algos without scaling based on gradient information really struggle to break symmetry here - SGD gets no where and Nesterov Accelerated Gradient / Momentum exhibits oscillations until they build up velocity in the optimization direction. Algos that scale step size based on the gradient quickly break symmetry and begin descent.*

Just to give you an hint on where the community is headed with Deep Learning

DALL-E OpenAI (January 2021)



DALL-E OpenAI

OpenAI DALL-E - 12-billion parameters trained with self-supervision

Yikes! 12×10^9 floating points parameters to train

> DALL·E is a **12-billion parameter** version of GPT-3 trained to generate images from text descriptions, using a dataset of text–image pairs. We've found that it has a diverse set of capabilities, including creating anthropomorphized versions of animals and objects, combining unrelated concepts in plausible ways, rendering text, and applying transformations to existing images.

~~0) Quick Intro to Optimization in Deep Learning~~

1) Network Structure: Multi-Layer Perceptron (MLP) is a Fully-Connected Neural Net

2) Backpropagation

1) Network Structure: Multi-Layer Perceptron (MLP)

is a Fully-Connected Neural Net

Networks and Topics that we do NOT cover

You will meet them at **Deep Learning** course

- **Convolutional** Neural Nets (good for images or any matrix data like as input)
- Generative Adversarial Networks **(GAN)** and adversarial training
- AutoEncoders or Variational Autoencoders
- Adversarial Attacks to NN
- Recurrent Neural Nets (RNN such as GRU, LSTM)
- Transformer Networks

Let's go back to single layer, linear soft-max regression or linear neural network

Let's recall last classification layer of a neural net as pipeline

$$\mathbf{x} \implies \mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b} \implies e^{\mathbf{z}} \implies \mathbf{p} = \frac{e^{\mathbf{z}}}{\sum_k e^{\mathbf{z}}} \implies -\ln(\mathbf{p}_y)$$

Representation of a Single Layer

Let's consider our linear softmax regressor

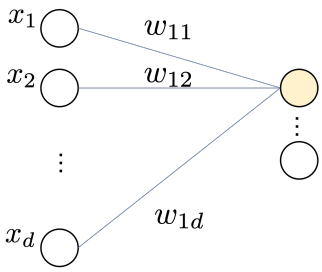
$$\underbrace{\mathbf{z}}_{\mathrm{R}^{K\mathrm{x}1}} = \underbrace{\mathbf{W}}_{\mathrm{R}^{K\times d}} \underbrace{\mathbf{x}}_{\mathrm{R}^{d\times 1}} + \underbrace{\mathbf{b}}_{\mathrm{R}^{K}}$$

We interpret as **Linear Layer** $\mathbf{W}\mathbf{x} + \boldsymbol{b}$ followed by **Non-Linear Activation function** σ

$$\sigma(\mathbf{W}\mathbf{x} + \boldsymbol{b}) = \sigma \circ \begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1d} \\ w_{21} & w_{22} & \cdots & w_{2d} \\ \vdots & \cdots & \ddots & \vdots \\ w_{k1} & w_{m2} & \cdots & w_{kd} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_k \end{pmatrix} = \sigma \circ \begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_k \end{pmatrix}$$

Representation of a Single Layer

$$\sigma(\mathbf{W}\mathbf{x} + \boldsymbol{b}) = \sigma \circ \begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1d} \\ - & - & & - \\ w_{21} & w_{22} & \cdots & w_{2d} \\ \vdots & \cdots & \ddots & \vdots \\ w_{k1} & w_{m2} & \cdots & w_{kd} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_k \end{pmatrix} = \sigma \circ \begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_k \end{pmatrix}$$



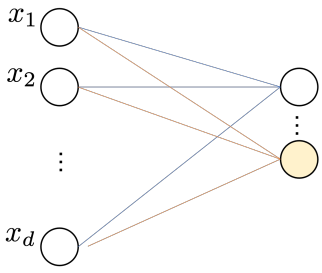
$$\mathbf{W}_1^T\mathbf{x} + b_1$$

Represent the "firing" for class k=1 *(bias is omitted from drawing)*

The row vector $\mathbf{W_1}$ **is a "neuron" or "unit"**
In this special case also is connected with
The weight vector of class k=1.

Representation of a Single Layer

$$\sigma(\mathbf{W}\mathbf{x} + \boldsymbol{b}) = \sigma \circ \begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1d} \\ w_{21} & w_{22} & \cdots & w_{2d} \\ \vdots & \cdots & \ddots & \vdots \\ w_{k1} & w_{m2} & \cdots & w_{kd} \\ - & - & & - \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_k \end{pmatrix} = \sigma \circ \begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_k \end{pmatrix}$$



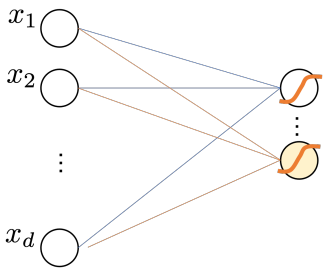
$$\mathbf{W}_k^T\mathbf{x} + b_k$$

Represent the "firing" for class k *(bias is omitted from drawing)*

The row vector $\mathbf{W_k}$ **is a "neuron" or "unit"**
In this special case also is connected with
The weight vector of class k.

Representation of a Single Layer: Linear plus non-Linear

$$\sigma(\mathbf{W}\mathbf{x}+\boldsymbol{b}) = \sigma \circ \begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1d} \\ w_{21} & w_{22} & \cdots & w_{2d} \\ \vdots & \cdots & \ddots & \vdots \\ w_{k1} & w_{m2} & \cdots & w_{kd} \\ - & - & & - \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_k \end{pmatrix} = \sigma \circ \begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_k \end{pmatrix}$$



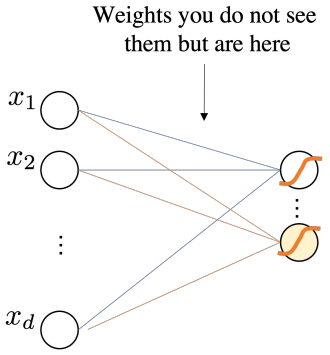
$$\sigma(\mathbf{W}_k^T\mathbf{x} + b_k)$$

Represent the "firing" for class k *(bias is omitted from drawing)*

Linearity is followed by non-linearity

Representation of a Single Layer: Linear plus non-Linear

$$\mathbf{W}\mathbf{x} = \begin{pmatrix} - & \text{unit} & - \\ & \vdots & \\ - & \text{unit} & - \end{pmatrix} \begin{pmatrix} | \\ \mathbf{x} \\ | \end{pmatrix}$$

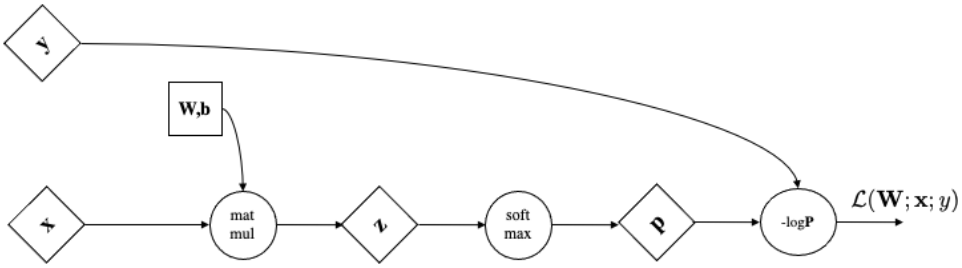


$$\sigma(\mathbf{W}_k^T\mathbf{x} + b_k)$$

Represent the "firing" for class k *(bias is omitted from drawing)*

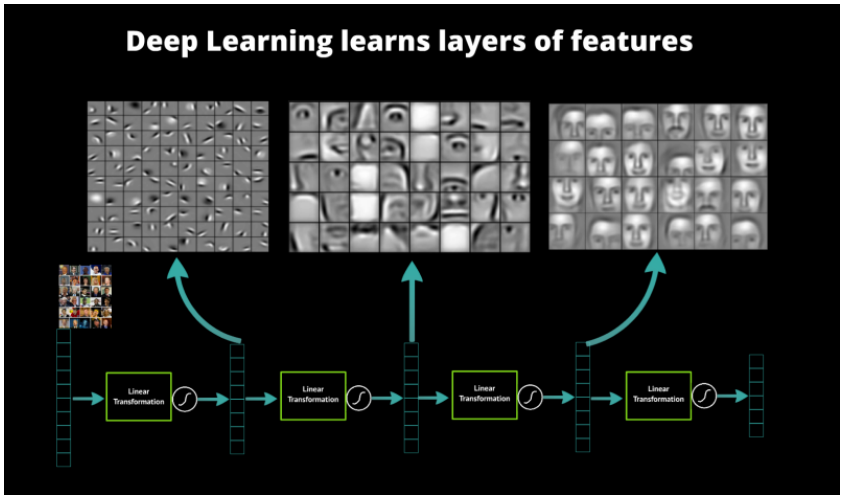
Linearity is followed by non-linearity

Representation as a computational graph



Damn, until now is all linear. So now the "Deep"!

- Damn, until now is all linear.
- Our **beloved SoftMax+CE linear layer** is there **in the end (classifier).**



Question: A Single Linear Soft-Max Layer may suffer from Bias or Variance problem?

A single linear layer is not enough for highly non-linear problems

Adding another non-linear layer before the classifier

- We improve the *expressiveness* of our learned function by adding another linear layer **before** the classification layer.
- Think this new layer as a feature map $\mathbf{x} \mapsto \phi(\mathbf{x})$; it maps our attribute to a feature space
- Now the classifier does not classify anymore directly $\mathbf{x}$ but the feature $\phi(x)$.
- Sorry, notation becomes complex. Upper script means layer index; lower-script selects the unit
- $\mathbf{W}^1 \in \mathrm{R}^{d \times p}$, $\boldsymbol{b}^1 \in \mathrm{R}^p$ so then $\mathbf{W}^2 \in \mathrm{R}^{p \times k}$, $\boldsymbol{b}^2 \in \mathrm{R}^k$

$$\mathbf{p} = \sigma(\mathbf{W}^2\Big(\underbrace{\sigma(\mathbf{W}^1\mathbf{x} + \boldsymbol{b}^1)}_{\boldsymbol{\phi}(x)}\Big) + \boldsymbol{b}^2)$$

dim. analysis: $d \mapsto p \mapsto k$

$\mathbf{W}^1 \in \mathrm{R}^{d \times p}$ is an Hidden Layer

Because it maps the original attribute in d from an dimensionality p and then p is used for classifying.

A priori you do not know what $\mathbf{W}^1$ may learn.

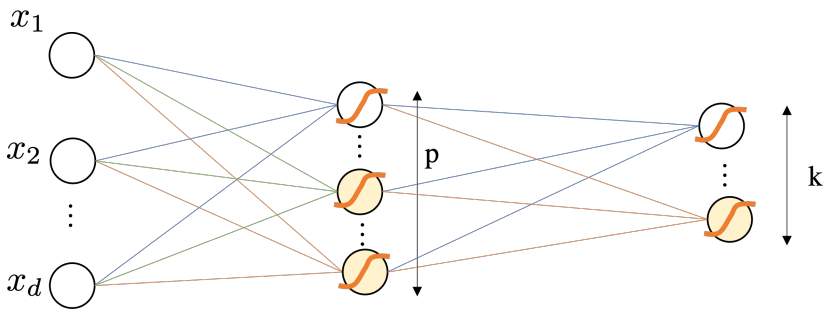
$$\mathbf{p} = \sigma(\mathbf{W}^2\underbrace{\Big(\sigma(\mathbf{W}^1\mathbf{x} + \boldsymbol{b}^1)\Big)}_{\phi(x)} + \boldsymbol{b}^2)$$

dim. analysis: $\quad d \mapsto p \mapsto k$

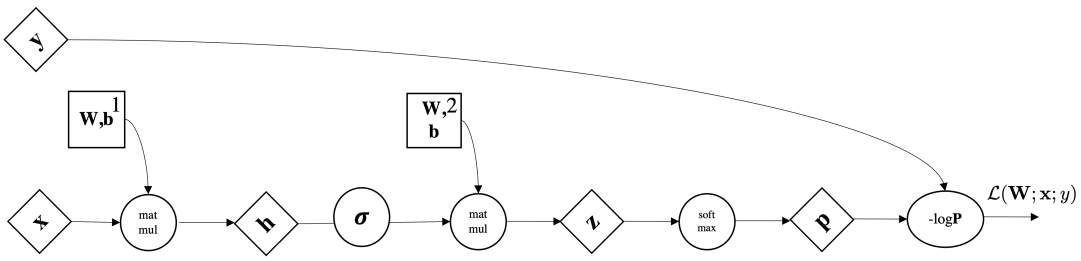
Let's update our visualizations

Multi-Layer Perceptron (MLP) with one hidden layer

Given the nature of these layers, they're called Fully-Connected NN



Multi-Layer Perceptron with one hidden layer



Non-linear activation functions: Sigmoid

Very important: **Activation Functions are computed element-wise.**

$$\sigma(z) = \frac{1}{1 + \exp^{-z}} \quad \text{sigmoid or logistic function}$$

Non-linear activation functions: Sigmoid

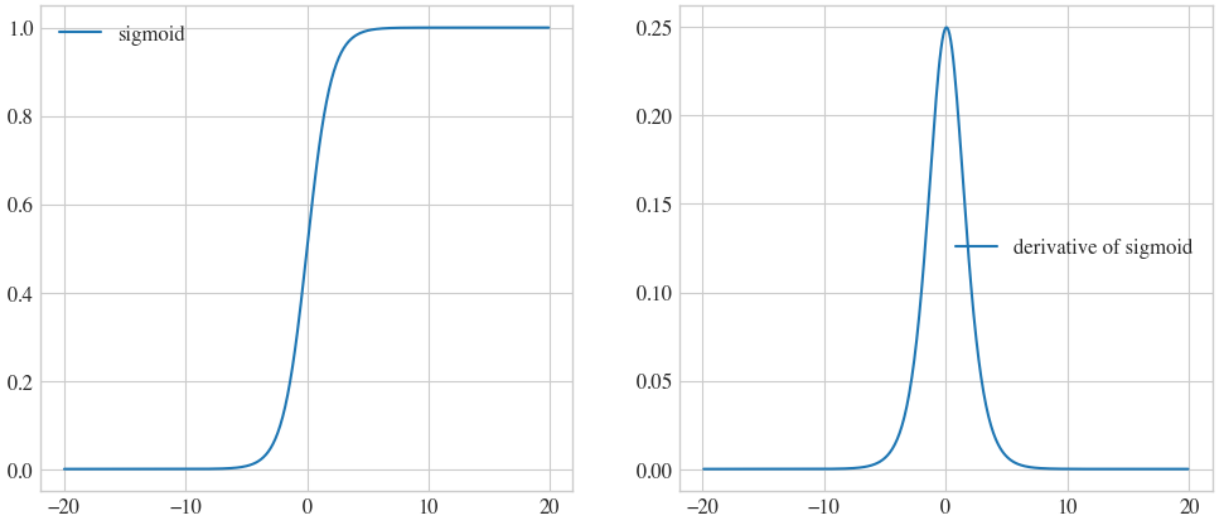
Very important: **Activation Functions are computed element-wise.**

$$\sigma(z) = \frac{1}{1 + \exp^{-z}} \quad \text{sigmoid or logistic function}$$

Smooth and Differentiable alternative to sign

{{import numpy as np; import matplotlib.pyplot as plt; step=0.1; x = np.arange(-20.0, 20.0, step); fig, axes = plt.subplots(1,2,figsize=(12,5)); y = 1/(1+np.exp(-x)); dy = np.diff(y); axes[0].plot(x,y); axes[0].legend(['sigmoid']); axes[1].plot(x[1:],dy/step); _=axes[1].legend(['derivative of sigmoid']);}}

In [4]:
```
import numpy as np;
import matplotlib.pyplot as plt; step=0.1;
x = np.arange(-20.0, 20.0, step);
fig, axes = plt.subplots(1,2,figsize=(12,5));
y = 1/(1+np.exp(-x));
dy = np.diff(y);
axes[0].plot(x,y)
axes[0].legend(['sigmoid'])
axes[1].plot(x[1:],dy/step);
axes[1].legend(['derivative of sigmoid']);
```



Non-linear activation functions: ReLu - Rectified Linear Unit

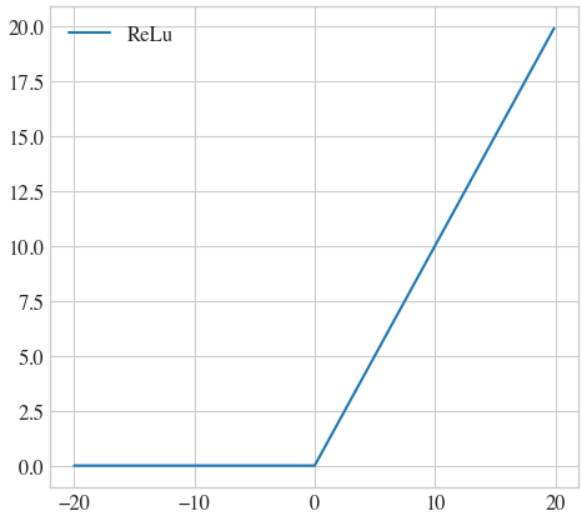
Very important: **Activation Functions are computed element-wise.**

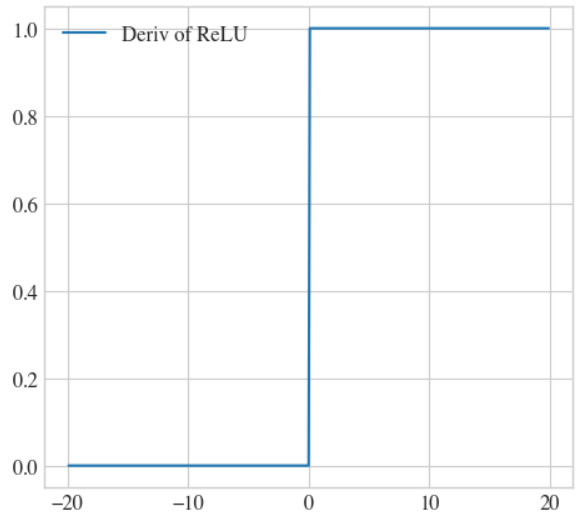
$$\sigma(z) = \max(0, z) \quad \text{ReLu}$$

ReLu is piece-wise linear function

{{import numpy as np; import matplotlib.pyplot as plt; step=0.1; x = np.arange(-20.0, 20.0, step); fig, axes = plt.subplots(1,2,figsize=(12,5)); y = np.maximum(0,x); dy = np.diff(y); axes[0].plot(x,y); axes[1].plot(x[1:],dy/step); axes[0].legend(['ReLu']); _=axes[1].legend(['Deriv of ReLU']);}}

In [5]:
```
import numpy as np;
import matplotlib.pyplot as plt; step=0.1;
x = np.arange(-20.0, 20.0, step);
fig, axes = plt.subplots(1,2,figsize=(12,5));
y = np.maximum(0,x);
dy = np.diff(y);
axes[0].plot(x,y)
axes[1].plot(x[1:],dy/step);
axes[0].legend(['ReLu'])
_=axes[1].legend(['Deriv of ReLU']);
```





Sigmoid

- Used to model output probability
- Nowadays not used in middle layers
- Have to compute $\exp()$
- **Vanishing gradients** for large input magnitude

ReLU

- Computationally efficient (no exp!)
- No vanishing gradients but do not let pass gradients for negative values
- Converges much faster than sigmoid (6x)
- Not differentiable in zero (subgradients)

There are other activation functions we do not cover

TanH, Leaky ReLU, parametrized ReLU, ELU

Exam question lookalike

How many **trainable parameters** do you have with a **MLP** with input features in R^d, a first layer with p units/neurons and another layer with k units/neurons that model the k classes for classification. Assume all layers have the bias term.

Exam question lookalike

We have two layers:

1. Maps d dimension to p with bias also. So we need to train $\underbrace{d \times p}_{weights} + \underbrace{p}_{bias}$ parameters
2. Maps p dimension to k with bias also. So we need to train $\underbrace{p \times k}_{weights} + \underbrace{k}_{bias}$ parameters

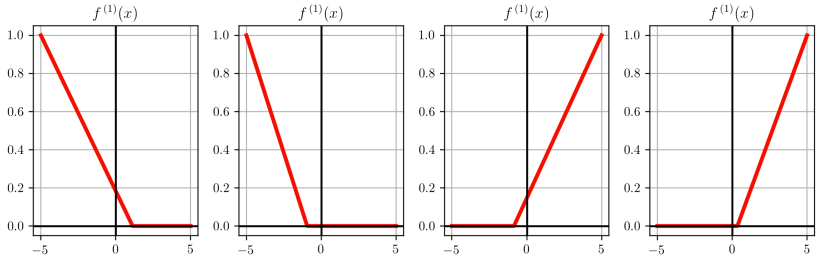
Total params = $(d \times p + p) + (p \times k + k)$

so if $d = 100$, $p = 2000$, and $k = 10$. **Total params = 222,010**

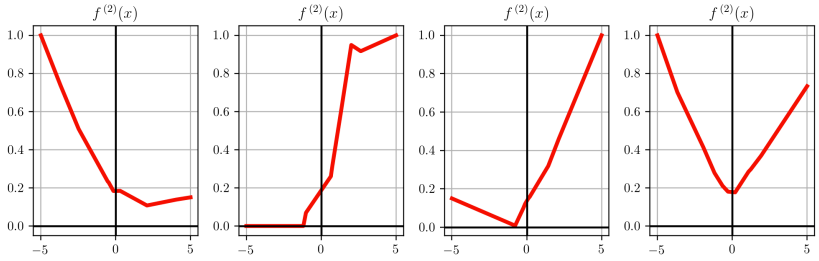
Adding Layers improves the expressiveness of the function

Example with ReLU

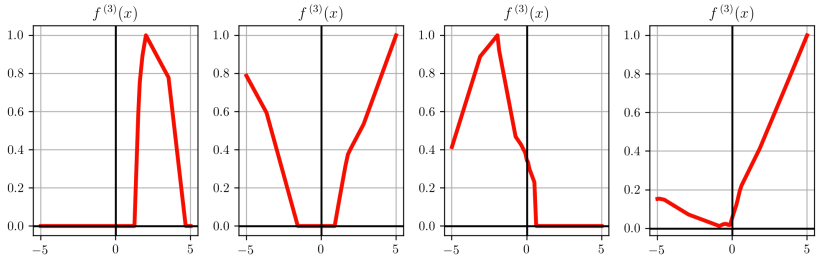
1 layers



2 layers



3 layers



Neural Net Initialization

Why we do not initialize all weights and biases to zero?

Consider σ as **Sigmoid activation function** everywhere, what happens if $\mathbf{W}$ are zeros?

No matter what is the input, the output will be always $\sigma(0) = 0.5$, in other words a **constant vector.**

$$\mathbf{z}^{[1]} = \mathbf{W}^{[1]}\mathbf{x}^{(i)} + \mathbf{b}^{[1]}$$
$$\mathbf{a}^{[1]} = \sigma\left(\mathbf{z}^{[1]}\right)$$
$$\mathbf{z}^{[2]} = \mathbf{W}^{[2]}\mathbf{a}^{[1]} + \mathbf{b}^{[2]}$$
$$\mathbf{a}^{[2]} = \sigma\left(\mathbf{z}^{[2]}\right)$$
$$\mathbf{z}^{[3]} = \mathbf{W}^{[3]}\mathbf{a}^{[2]} + \mathbf{b}^{[3]}$$
$$\hat{y}^{(i)} = \mathbf{a}^{[3]} = \sigma\left(\mathbf{z}^{[3]}\right)$$

OK, so we will initialize the weights to a constant value different than zero? Will this work?

NO, it won't!

Each element of the activation vector will be a scaled version of the input. This behavior will occur at all layers of the neural network. As a result, when we compute the gradient, all neurons in a layer will be equally responsible for anything contributed to the final loss. **We call this property symmetry.**

This means each neuron (within a layer) will receive the **exact same gradient update value (i.e., all neurons will learn the same thing).**

Random Asymmetric Initialization is needed

Randomness breaks the symmetric so that each neuron learns different things (gradients updates are different).

$$\mathbf{W}_{ij} \sim \mathrm{U}(-0.1, 0.1)$$

$$\mathbf{W}_{ij} \sim \mathrm{N}(0, 0.01)$$

Xavier/He Initialization

$$\mathbf{W}_{ij}^{l} \sim \mathrm{N}\left(0, \sqrt{\frac{2}{n^{l} + n^{l-1}}}\right)$$

- n^{l} is the **number of neurons** in the layer l-th
- n^{l-1} is the **number of neurons** in the layer $l-1$-th (previous layer to l).

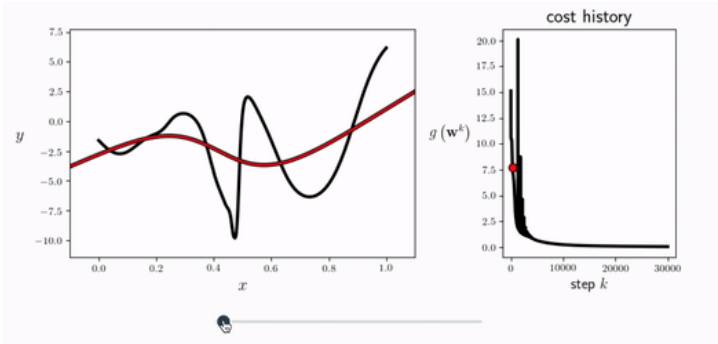
Idea: For a single layer, consider the variance of the input to the layer as σ^{in} and the variance of the output (i.e., activations) of a layer to be σ^{out}. Xavier/He initialization encourages σ^{in} to be similar to σ^{out}.

We can have more than 3 layers

In principle, we can compose how many layers we want.

Think of each layer has **learning a feature map on top of the previous feature map and then you have your final layer for your task at hand (classification, regression)**

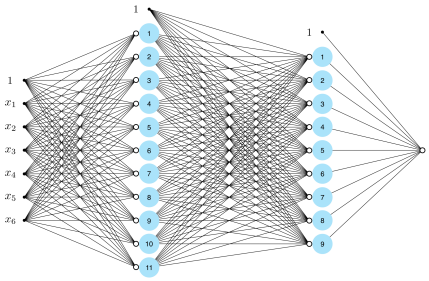
$$\mathbf{y} = f \circ (\sigma \circ f) \circ \cdots \circ (\sigma \circ f)(\mathbf{x})$$



Universal Approximation Theorem [Informal]

Given a continuous function $\mathbf{y} = f(\mathbf{x})$ where $\mathbf{x} \in \mathrm{R}^d$ and $\mathbf{y} \in \mathrm{R}^k$, considering only a bounded region of $\mathbf{x}$, **there exists** a single-hidden-layer NN_θ with a **finite number of neurons/units in the hidden layer**, such that:

$$|f(\mathbf{x}) - NN_\theta(\mathbf{x})| \leq \epsilon$$



1. We do not know how to reach that $NN_\theta(\mathbf{x})$
2. It does not tell you how many samples you need to recover the approximation.

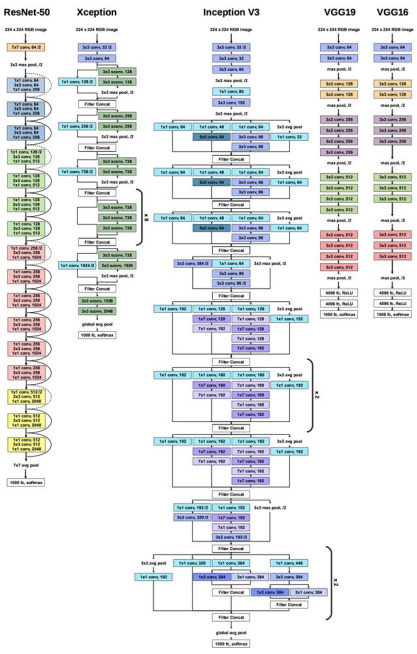
~~0) Quick Intro to Optimization in Deep Learning~~

~~1) Network Structure: Multi-Layer Perceptron (MLP) is a Fully-Connected Neural Net~~

2) Backpropagation

Backpropagation and Differential Programming

NN can be huge composition of functions! 😱



Three ways of computing the gradients $\nabla_{\mathbf{W}} \mathrm{L}(x, y; \mathbf{w})$

1. **Manually** (if we change the network, we have to adjust it for a 100 layer neural net) maybe not a good idea, does not scale, even if we use symbolic derivation tools such as Mathematica ✍🏻
2. **Finite Difference** good to check the gradients once you have an automatic way of computing it; **very slow, unfeasible in training!** 👩🏽
3. **Backpropagation**: application of chain rule of calculus to tensors with a computational graph with caching **(differential programming with automatic differentiation)** 🖥

Infeasible to derive manully the gradient update

Finite difference (very slow!) but used for gradient check

- It could be used to check if your backpropagation chain is correct
- It is not used when you train; computationally expensive

Implement the definition of derivative and apply it numerically on vectors:

$$\frac{df}{dx}(x) = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon}.$$

Finite difference (very slow!) but used for gradient check

How do we read this formula for finite difference with neural net?

Assume $\mathbf{W}$ is your matrix and $w = \mathbf{W}_{ij}$ is a scalar inside your matrix.

For `i,j=1....dims`:

1. $w = \mathbf{W}_{ij}$
2. **Evaluate your NN loss at current weight value** `L(w)`
3. You want to see what is the impact of a paramter w on the loss?
4. Perturb that w by an $\epsilon = 1e-5$ and evaluate the new loss at `L(w+eps)`
5. **Numerical Gradient is** `[L(w+eps)-L(w)]/eps` at position ij, so store it in $\nabla_{\mathbf{W}} \mathrm{L}_{ij}$

$$\frac{\partial \mathrm{L}}{\partial w}(x, y; w) = \frac{L(w+\epsilon) - L(w)}{\epsilon}$$

At the end you have your **numerical gradients** $\nabla_{\mathbf{W}} \mathrm{L}_{ij}$.

Backpropagation

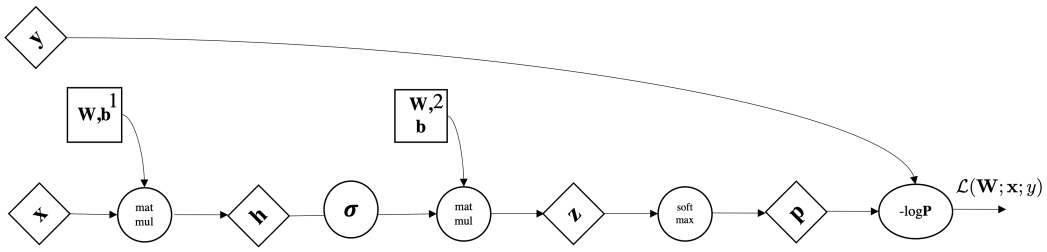
Who invented Backpropagation

https://people.idsia.ch/~juergen/who-invented-backpropagation.html

Let's be clear on what we need to compute

$\forall l \in [1 \ldots, L]$:

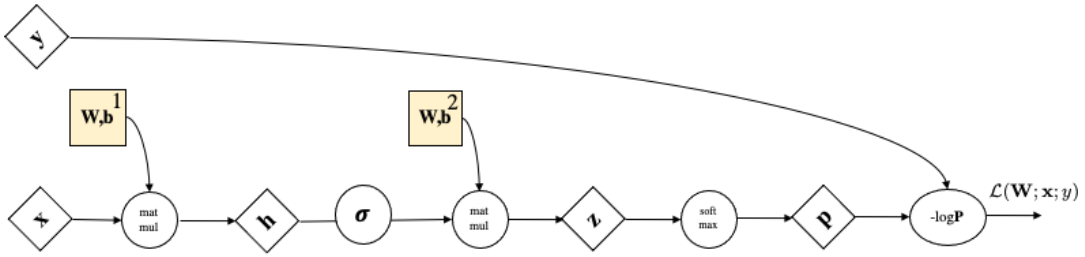
1. $\nabla_{\mathbf{W}^l} \mathrm{L}(\mathbf{x}, y; \{\mathbf{W}, b\})$
2. $\nabla_{\mathbf{b}^l} \mathrm{L}(\mathbf{x}, y; \{\mathbf{W}, b\})$



Once you have gradients on ALL weights $\Longrightarrow$ We can update

$\forall l \in [1 \ldots, L]$:

1. $\mathbf{W}^l \leftarrow \mathbf{W}^l - \gamma \nabla_{\mathbf{W}^l} \mathrm{L}(\mathbf{x}, y; \{\mathbf{W}, b\})$
2. $\mathbf{b}^l \leftarrow \mathbf{b}^l - \gamma \nabla_{\mathbf{b}^l} \mathrm{L}(\mathbf{x}, y; \{\mathbf{W}, b\})$



How do we get all the weights?

Mostly taken from here

Chain Rule

Returning to functions of a single variable, suppose that $y = f(g(x))$ and that the underlying functions $y = f(u)$ and $u = g(x)$ are both differentiable. The chain rule states that

$$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx}.$$

What is the **derivative of loss wrt x in the equation below**?

$$y = loss\big(g(h(i(x)))\big)$$

$$\frac{\partial loss}{\partial x} = \frac{\partial loss}{\partial g}\frac{\partial g}{\partial h}\frac{\partial h}{\partial i}\frac{\partial i}{\partial x}$$

Autodiff

pytorch = numpy + autodiff

In pytorch this mechanism is called **autograd**

Let's consider quadratic form

$$f(\mathbf{x}) = \mathbf{x}^\top \mathbf{A}\mathbf{x} + \mathbf{b}^\top \mathbf{x} + c$$

Let's consider quadratic form

$$f_{\mathbf{A},\mathbf{b},c}(\mathbf{x}) = \mathbf{x}^\top \mathbf{A}\mathbf{x} + \mathbf{b}^\top \mathbf{x} + c$$

Let's consider quadratic form

Can be viewed as a function of:

- input $\mathbf{x}$; fixing $\theta \rightarrow$ *Used when we study how our model reacts to perturbation of input*
- input θ; fixing $\mathbf{x} \rightarrow$ (training set is given, find the weights) *Used when we learn*

$$f(\mathbf{x}; \theta) = \mathbf{x}^\top \mathbf{A}\mathbf{x} + \mathbf{b}^\top \mathbf{x} + c \qquad \text{where} \quad \theta = \mathbf{A}, \mathbf{b}, c$$

Let's consider quadratic form

$$f(\theta; \mathbf{x}) = \mathbf{x}^\top \mathbf{A}\mathbf{x} + \mathbf{b}^\top \mathbf{x} + c \qquad \text{where} \quad \theta = \mathbf{A}, \mathbf{b}, c \text{ are fixed}$$

Let us code this function with Pytorch

In [6]:
```
import torch
import random
random.seed(0) # to fix the random seed to make the code deterministic
torch.manual_seed(0) # to fix the random seed to make the code deterministic
print(torch.__version__)
```

```
2.1.2
```

We assume the parameters θ as given and fix and we optimize the input $\mathbf{x}$.

We assume $\mathbf{x} \in \mathrm{R}^2$.

In [7]:
```
## params
A = torch.tensor([[0., 2.], [2.,0.]]) # 2x2
b = torch.tensor([[-0.5],[ 0.5]])  # 2x1
c = torch.tensor([-2.], dtype=torch.float32) # 1x1
## input
x = torch.tensor([1., 1.], requires_grad=True) # 1x2 Note how here we require the gradients
thetas = (A, b, c) # pack the parameters
```

In [8]:
```
A.shape, b.shape, c.shape, x.shape
```

Out[8]:
```
(torch.Size([2, 2]), torch.Size([2, 1]), torch.Size([1]), torch.Size([2]))
```

In [9]:
```
thetas
```

Out[9]:
```
(tensor([[0., 2.],
         [2., 0.]]),
 tensor([[-0.5000],
         [ 0.5000]]),
 tensor([-2.]))
```

Let's edit it a bit:

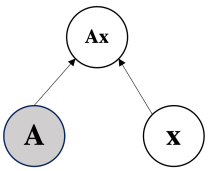
$$f(\theta; \mathbf{x}) = \left(\mathbf{x}^\top \mathbf{A}\mathbf{x} + \mathbf{b}^\top \mathbf{x} + c\right)^2 \qquad \text{where} \quad \theta = \mathbf{A}, \mathbf{b}, c \text{ are fixed}$$

Code in pytorch

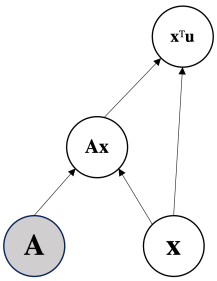
In [10]:
```
def function(x, thetas):
    A, b, c = thetas
    return torch.pow(x.T @ A @ x + b.T@x + c, 2)
```

What pytorch does under the hood

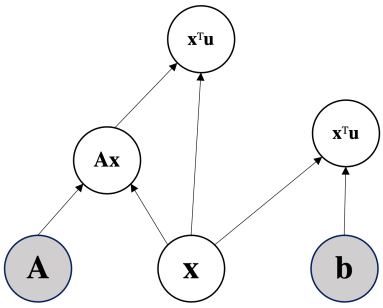
$$f_{\mathbf{A},\mathbf{b},c}(\mathbf{x}) = \quad \mathbf{Ax}$$



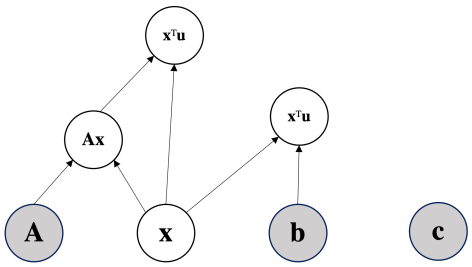
$$f_{\mathbf{A},\mathbf{b},c}(\mathbf{x}) = \mathbf{x}^\top \mathbf{Ax}$$



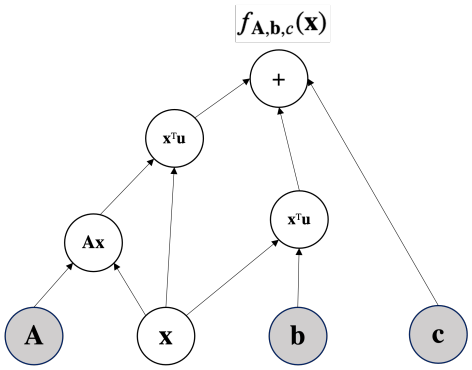
$$f_{\mathbf{A},\mathbf{b},c}(\mathbf{x}) = \mathbf{x}^\top \mathbf{Ax} \quad \mathbf{b}^\top \mathbf{x}$$



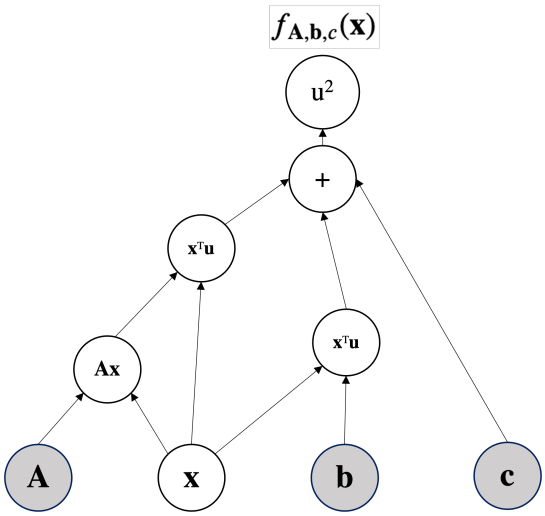
$$f_{\mathbf{A},\mathbf{b},c}(\mathbf{x}) = \mathbf{x}^\top \mathbf{A}\mathbf{x} \quad \mathbf{b}^\top \mathbf{x} \quad c$$



$$f_{\mathbf{A},\mathbf{b},c}(\mathbf{x}) = \mathbf{x}^\top \mathbf{A}\mathbf{x} + \mathbf{b}^\top \mathbf{x} + c$$



$$f(\theta; \mathbf{x}) = \left(\mathbf{x}^\top \mathbf{A}\mathbf{x} + \mathbf{b}^\top \mathbf{x} + c\right)^2$$



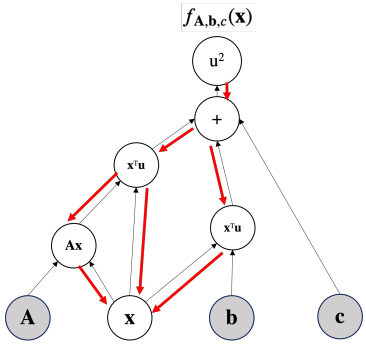
Optimize $\mathbf{x}$ with SGD

In [11]:
```
for epoch in range(200):
    if x.grad is not None: x.grad.zero_() # zeroing the gradient before backprop
    func_v = function(x, thetas) # eval function and track gradients (autograd)
    func_v.backward() # get the grads
    if epoch % 20 == 0:
        print(f'{epoch}) y before SGD = ',func_v.item(),'input x=',x.detach().numpy())
    with torch.no_grad(): # here we DO not track gradient
        x += -1e-3*x.grad # plain SGD
        if epoch % 20 == 0:
            print(f'{epoch}) y after SGD  = ',function(x,thetas).item(),
                  'input x=',x.detach().numpy())
```

```
/var/folders/rt/lg7n4lt1489270pz_18qn1_c0000gp/T/ipykernel_61746/880405579.py:3: UserWarning: The use of `x.T`
on tensors of dimension other than 2 to reverse their shape is deprecated and it will throw an error in a futur
e release. Consider `x.mT` to transpose batches of matrices or `x.permute(*torch.arange(x.ndim - 1, -1, -1))` t
o reverse the dimensions of a tensor. (Triggered internally at /Users/runner/work/pytorch/pytorch/pytorch/aten/
src/ATen/native/TensorShape.cpp:3618.)
  return torch.pow(x.T @ A @ x + b.T@x + c, 2)
0) y before SGD =  4.0 input x= [1. 1.]
0) y after SGD  =  3.5006706714630127 input x= [0.986 0.982]
20) y before SGD =  0.4590446949005127 input x= [0.84928155 0.79601455]
20) y after SGD  =  0.4189494550228119 input x= [0.8456445 0.7907337]
40) y before SGD =  0.08419127017259598 input x= [0.802158  0.7256682]
40) y after SGD  =  0.07775045186281204 input x= [0.80076367 0.72351605]
60) y before SGD =  0.018030276522040367 input x= [0.78288543 0.6954951 ]
60) y after SGD  =  0.01672857254743576 input x= [0.7822726 0.6945199]
80) y before SGD =  0.0041193654760718346 input x= [0.77417433 0.68153346]
80) y after SGD  =  0.0038299155421555042 input x= [0.7738886  0.68107176]
100) y before SGD =  0.000969333981629461 input x= [0.7700594 0.6748613]
100) y after SGD  =  0.0009021023870445788 input x= [0.76992244 0.6746384 ]
120) y before SGD =  0.0002313051954843104 input x= [0.7680748 0.6716252]
120) y after SGD  =  0.00021535974519792944 input x= [0.7680083  0.67151654]
140) y before SGD =  5.5571563279954717e-05 input x= [0.767108  0.6700446]
140) y after SGD  =  5.1754243031609803e-05 input x= [0.76707554 0.6699914 ]
160) y before SGD =  1.3398824194155168e-05 input x= [0.76663494 0.6692699 ]
160) y after SGD  =  1.2477918062359095e-05 input x= [0.766619  0.6692438]
180) y before SGD =  3.2350690162274987e-06 input x= [0.76640296 0.66888946]
180) y after SGD  =  3.0142657578835497e-06 input x= [0.76639515 0.66887665]
```

Gradient Flow

$$f(\theta; \mathbf{x}) = \left(\mathbf{x}^\top \mathbf{A}\mathbf{x} + \mathbf{b}^\top \mathbf{x} + c\right)^2$$



In [12]:
```
function(x, thetas).item()
```

Out[12]:
```
7.811336786289758e-07
```

In [13]:
```
x.T @ A @ x + b.T@x + c
```

Out[13]:
```
tensor([0.0009], grad_fn=<AddBackward0>)
```

In [14]:
```
x
```

Out[14]:
```
tensor([0.7663, 0.6687], requires_grad=True)
```

What we have found

The point $\mathbf{x} = [0.7815, 0.7137]$ satisfies approximatively $\mathbf{x}^\top \mathbf{A}\mathbf{x} + \mathbf{b}^\top \mathbf{x} + c \approx 0 \quad (0.1971)$

torch.no_grad()

```
with torch.no_grad(): # we do not track the backward pass
    code that uses the function...
```

In the DAG we only **do topological sorting forwarding** (no backward).

In [15]:
```
with torch.no_grad():
    f = function(x, thetas)
print(f, '\n\n-->if you see in the tensor, there is NO grad_fn=<AddBackward0>) as before')
```

```
tensor([7.8113e-07])

-->if you see in the tensor, there is NO grad_fn=<AddBackward0>) as before
```

Fine-tuning (freezing parameters in an NN)

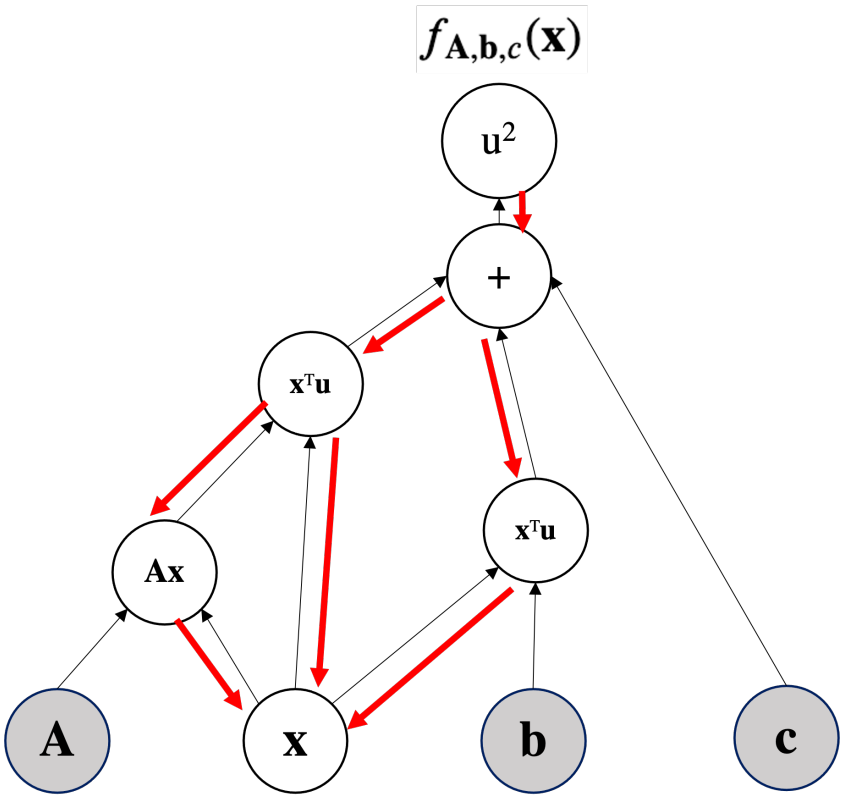
```
from torch import nn, optim

model = resnet18(weights=ResNet18_Weights.DEFAULT)

# Freeze all the parameters in the network
for param in model.parameters():
    param.requires_grad = False
```

torch.detach()

In [16]:
```
def function(x, thetas):
    A, b, c = thetas
    quad = x.T @ A @ x
    linear_detached = (b.T@x).detach() # detaching b*x
    return torch.pow( quad + linear_detached + c, 2)
```

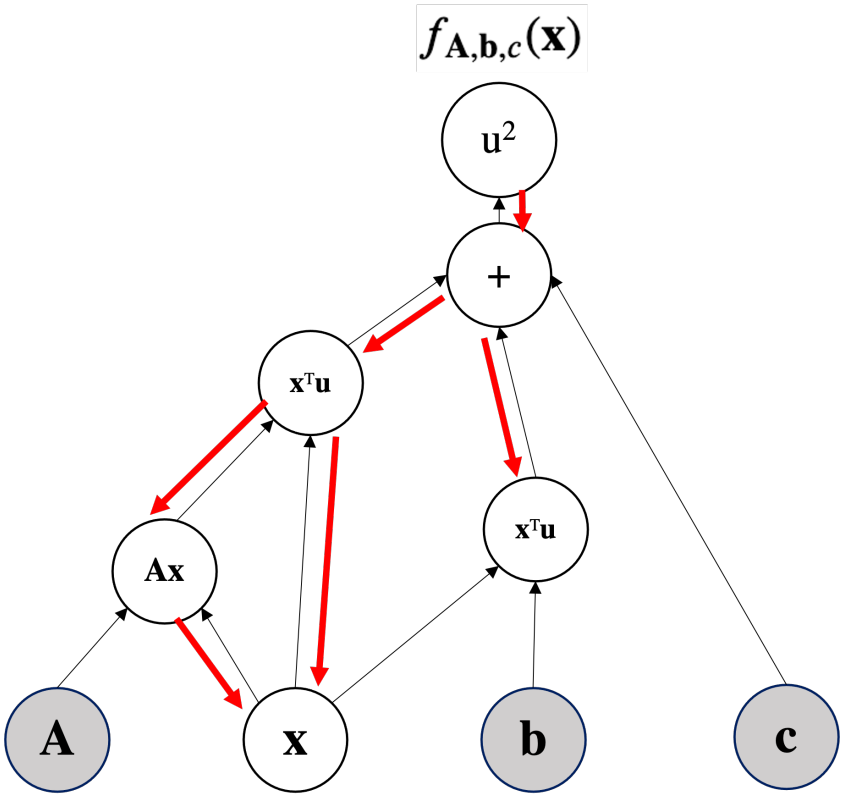
Gradient Flow

$$f(\theta; \mathbf{x}) = \left(\mathbf{x}^\top \mathbf{A}\mathbf{x} + \mathbf{b}^\top \mathbf{x} + c\right)^2$$



Gradient Flow with Detach

$$f(\theta; \mathbf{x}) = \left(\mathbf{x}^\top \mathbf{A}\mathbf{x} + \mathbf{b}^\top \mathbf{x} + c\right)^2$$



Chain Rule on Directed Acyclic Graph (DAG)

Let us make an easier example

Automate the computation of derivatives with computer science

$$\mathrm{L}(x, y, z) = (x + y)z$$

$$x = -2;\ y = 5;\ z = -4;$$

The high school way

$$\frac{\partial \mathrm{L}(x, y, z)}{\partial x} = ?$$

$$\mathrm{L}(x, y, z) = (x + y)z$$

$$x = -2;\ y = 5;\ z = -4;$$

The high school way (as we did until now):

$$\frac{\partial \mathrm{L}(x, y, z)}{\partial x} = (\mathbf{x}z + yz)' = (\mathbf{x}z)' + (yz)' = z$$

$$\frac{\partial \mathrm{L}(x, y, z)}{\partial y} = (xz + \mathbf{y}z)' = (xz)' + (\mathbf{y}z)' = z$$

$$\frac{\partial \mathrm{L}(x, y, z)}{\partial z} = x + y$$

Now with Chain Rule

$$L(x, y, z) = (x + y)z$$

$$x = -2;\ y = 5;\ z = -4;$$

but we can re-write it with the chain rule:

$$\frac{\partial L(x, y, z)}{\partial x} = \big((\underbrace{x + y}_{q})z\big)' = \frac{\partial L}{\partial q}\frac{\partial q}{\partial x}$$

Now with Chain Rule

$$L(x, y, z) = (x + y)z$$

$$x = -2;\ y = 5;\ z = -4;$$

but we can re-write it with the chain rule:

$$\frac{\partial L(x, y, z)}{\partial x} = \big((\underbrace{x + y}_{q})z\big)' = \frac{\partial L}{\partial q}\frac{\partial q}{\partial x} = z\frac{\partial q}{\partial x}$$

Now with Chain Rule

$$L(x, y, z) = (x + y)z$$

$$x = -2;\ y = 5;\ z = -4;$$

but we can re-write it with the chain rule:

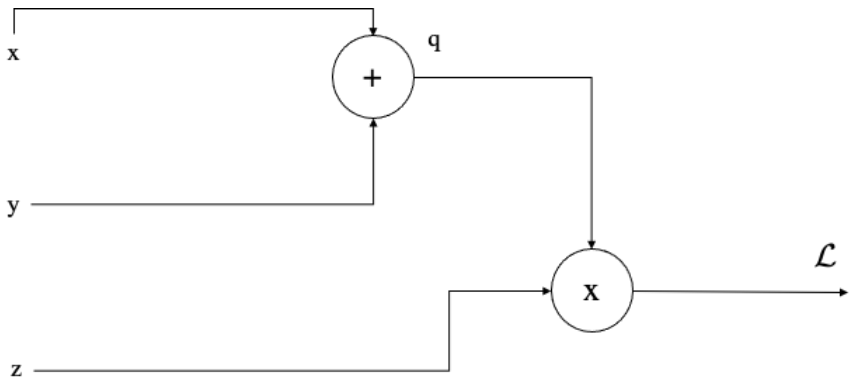
$$\frac{\partial L(x, y, z)}{\partial x} = \big((\underbrace{x + y}_{q})z\big)' = \frac{\partial L}{\partial q}\frac{\partial q}{\partial x} = z \cdot 1 = z$$

Chain Rule on Directed Acyclic Graph (DAG)

$$L(x, y, z) = (x + y)z$$

$$x = -2;\ y = 5;\ z = -4;$$

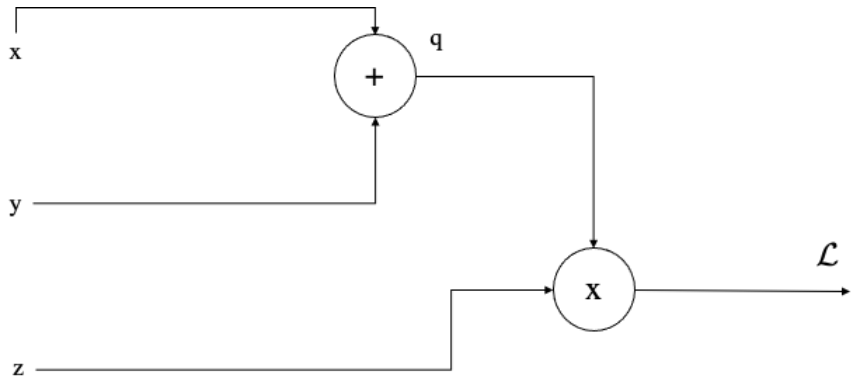
The computer science way:



$$\frac{\partial L}{\partial x}, \frac{\partial L}{\partial y}, \frac{\partial L}{\partial z}$$

Chain Rule on Directed Acyclic Graph (DAG)

Even if the problem is very small, we break it down to subproblems so that we can **automate** it:



$$\frac{\partial \mathrm{L}}{\partial x}, \frac{\partial \mathrm{L}}{\partial y}, \frac{\partial \mathrm{L}}{\partial z}$$

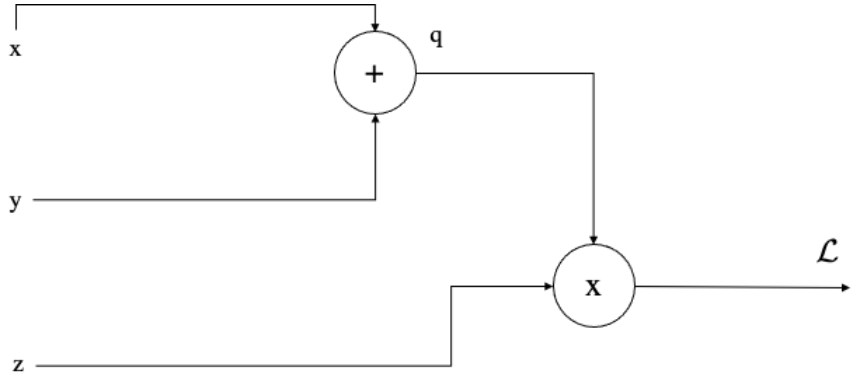
Who is input of L?

q and z are input of L.

$$\frac{\partial \mathrm{L}}{\partial q}, \frac{\partial \mathrm{L}}{\partial z}$$

Chain Rule on Directed Acyclic Graph (DAG)

Even if the problem is very small, we break it down to subproblems so that we can **automate** it:



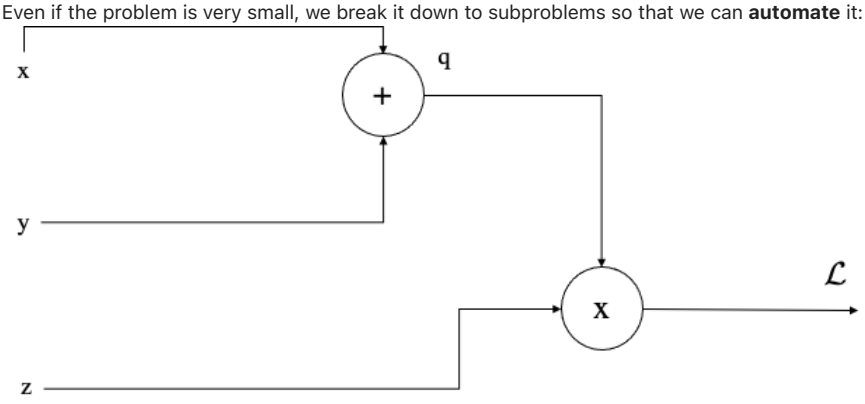
$$\frac{\partial \mathrm{L}}{\partial x}, \frac{\partial \mathrm{L}}{\partial y}, \frac{\partial \mathrm{L}}{\partial z}$$

What is the derivate? (Just check the operation at the gate)

$$\frac{\partial \mathrm{L}}{\partial q}, \frac{\partial \mathrm{L}}{\partial z}$$

Chain Rule on Directed Acyclic Graph (DAG)

Even if the problem is very small, we break it down to subproblems so that we can **automate** it:



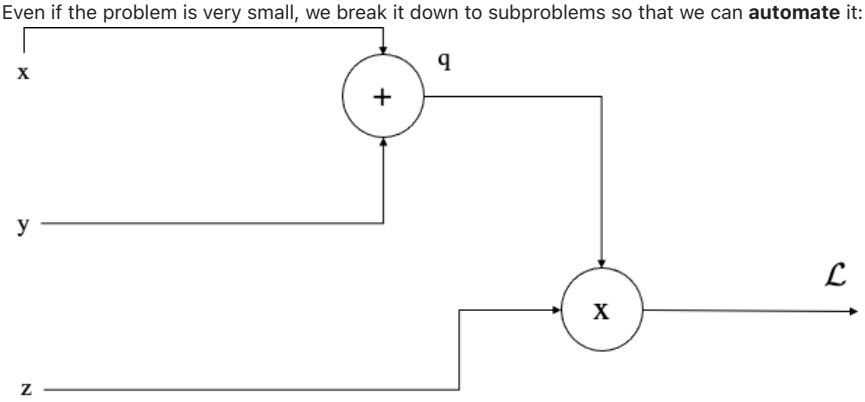
$$\frac{\partial \mathrm{L}}{\partial x}, \frac{\partial \mathrm{L}}{\partial y}, \frac{\partial \mathrm{L}}{\partial z}$$

What is the derivate? (Just check the operation at the gate)?

$$\frac{\partial \mathrm{L}}{\partial q} = z, \frac{\partial \mathrm{L}}{\partial z} = q$$

Chain Rule on Directed Acyclic Graph (DAG)

Even if the problem is very small, we break it down to subproblems so that we can **automate** it:



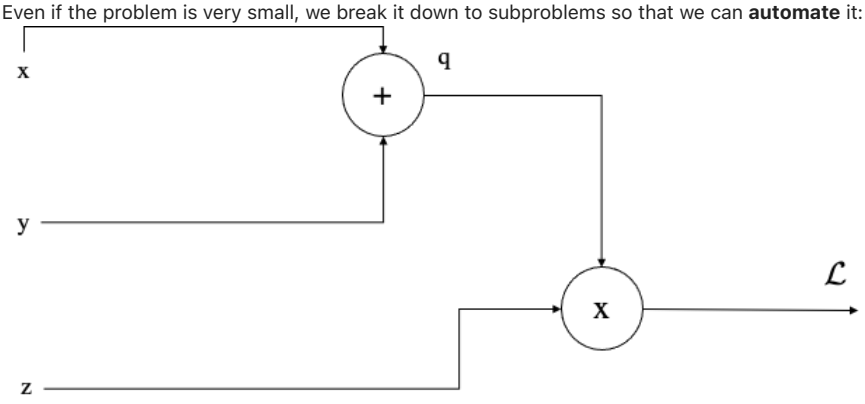
$$\frac{\partial \mathrm{L}}{\partial x}, \frac{\partial \mathrm{L}}{\partial y}, \frac{\partial \mathrm{L}}{\partial z}$$

What is the derivative? (Just check the operation at the gate)?

$$\frac{\partial \mathrm{q}}{\partial x} = 1, \frac{\partial \mathrm{q}}{\partial y} = 1$$

Chain Rule on Directed Acyclic Graph (DAG)

Even if the problem is very small, we break it down to subproblems so that we can **automate** it:

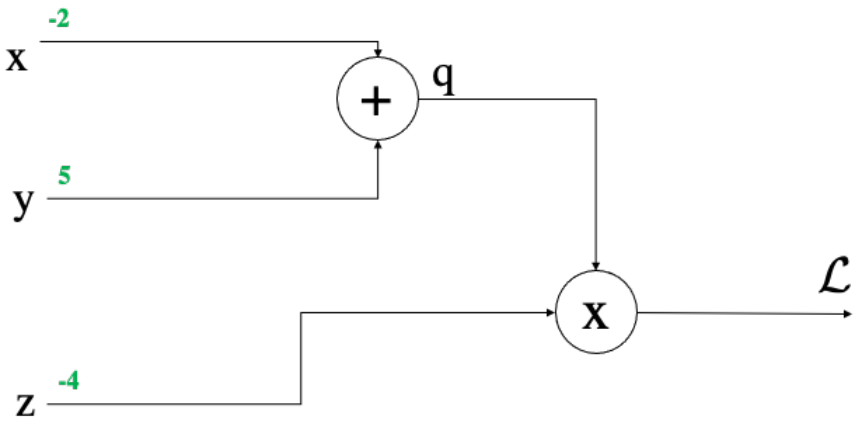


$$\frac{\partial \mathrm{L}}{\partial x}, \frac{\partial \mathrm{L}}{\partial y}, \frac{\partial \mathrm{L}}{\partial z}$$

OK now we have all the **analytical "local" partial derivatives, we can compute something**

$$\frac{\partial \mathrm{L}}{\partial q} = z, \frac{\partial \mathrm{L}}{\partial z} = q, \frac{\partial \mathrm{q}}{\partial x} = 1, \frac{\partial \mathrm{q}}{\partial y} = 1$$

Forward Pass

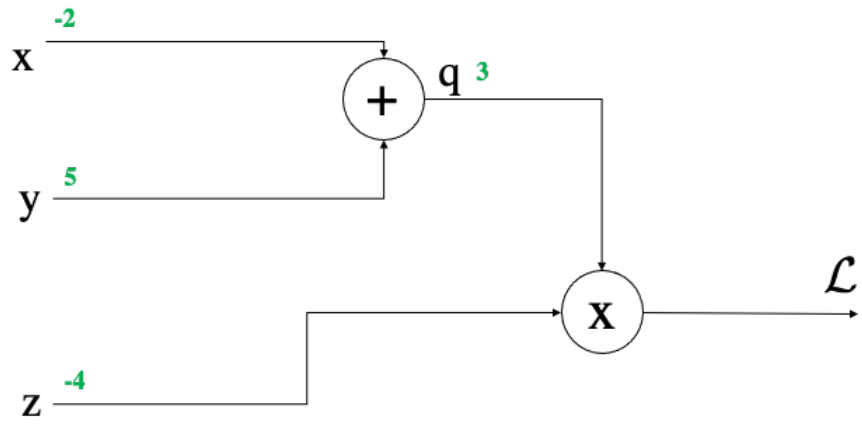


$$\frac{\partial \mathrm{L}}{\partial x}, \frac{\partial \mathrm{L}}{\partial y}, \frac{\partial \mathrm{L}}{\partial z}$$

OK now we have all the **analytical partial derivatives, we can compute something**

$$\frac{\partial \mathrm{L}}{\partial q} = z, \frac{\partial \mathrm{L}}{\partial z} = q, \frac{\partial \mathrm{q}}{\partial x} = 1, \frac{\partial \mathrm{q}}{\partial y} = 1$$

Forward Pass

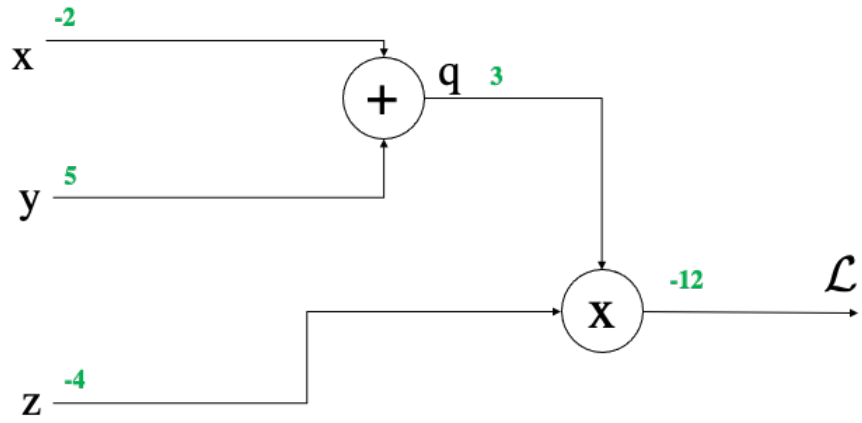


$$\frac{\partial \mathrm{L}}{\partial x}, \frac{\partial \mathrm{L}}{\partial y}, \frac{\partial \mathrm{L}}{\partial z}$$

OK now we have all the **analytical partial derivatives, we can compute something**

$$\frac{\partial \mathrm{L}}{\partial q} = z, \frac{\partial \mathrm{L}}{\partial z} = q, \frac{\partial \mathrm{q}}{\partial x} = 1, \frac{\partial \mathrm{q}}{\partial y} = 1$$

Forward Pass



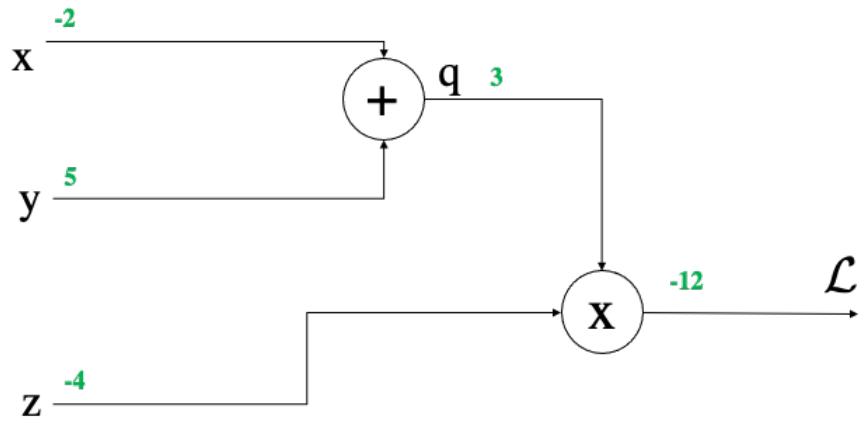
This is what we wanted:

$$\frac{\partial \mathrm{L}}{\partial x}, \frac{\partial \mathrm{L}}{\partial y}, \frac{\partial \mathrm{L}}{\partial z}$$

This is what we have

$$\frac{\partial \mathrm{L}}{\partial q} = z, \frac{\partial \mathrm{L}}{\partial z} = q, \frac{\partial \mathrm{q}}{\partial x} = 1, \frac{\partial \mathrm{q}}{\partial y} = 1$$

Backward Pass



Act as a base case for the recursion:

$$\frac{\partial \mathrm{L}}{\partial \mathrm{L}} = ?$$

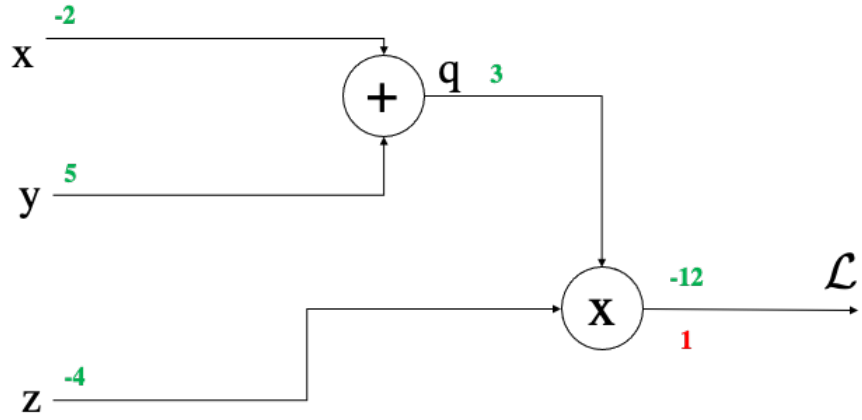
This is what we wanted:

$$\frac{\partial \mathrm{L}}{\partial x}, \frac{\partial \mathrm{L}}{\partial y}, \frac{\partial \mathrm{L}}{\partial z}$$

This is what we have

$$\frac{\partial \mathrm{L}}{\partial q} = z, \frac{\partial \mathrm{L}}{\partial z} = q, \frac{\partial \mathrm{q}}{\partial x} = 1, \frac{\partial \mathrm{q}}{\partial y} = 1$$

Backward Pass



Act as a base case for the recursion:

$$\frac{\partial \mathrm{L}}{\partial \mathrm{L}} = 1$$

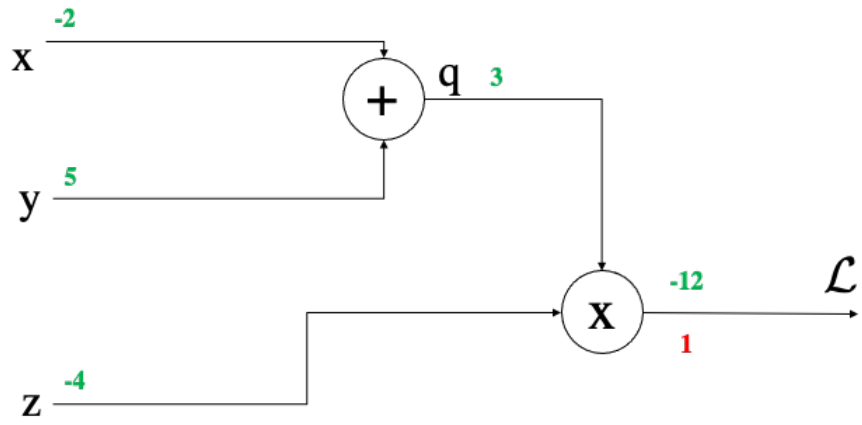
This is what we wanted:

$$\frac{\partial \mathrm{L}}{\partial x}, \frac{\partial \mathrm{L}}{\partial y}, \frac{\partial \mathrm{L}}{\partial z}$$

This is what we have

$$\frac{\partial \mathrm{L}}{\partial q} = z, \frac{\partial \mathrm{L}}{\partial z} = q, \frac{\partial \mathrm{q}}{\partial x} = 1, \frac{\partial \mathrm{q}}{\partial y} = 1$$

Backward Pass



What is the value of the gradient of L on z?:

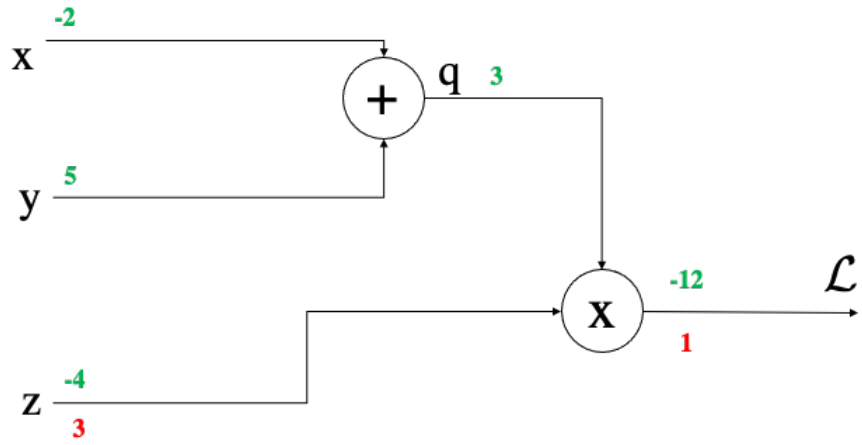
This is what we wanted:

$$\frac{\partial \mathrm{L}}{\partial x}, \frac{\partial \mathrm{L}}{\partial y}, \frac{\partial \mathrm{L}}{\partial z}$$

This is what we have

$$\frac{\partial \mathrm{L}}{\partial q} = z, \frac{\partial \mathrm{L}}{\partial z} = q, \frac{\partial \mathrm{q}}{\partial x} = 1, \frac{\partial \mathrm{q}}{\partial y} = 1$$

Backward Pass



What is the value of the gradient of L on z?:

$$\frac{\partial \mathrm{L}}{\partial z} = q = 3$$

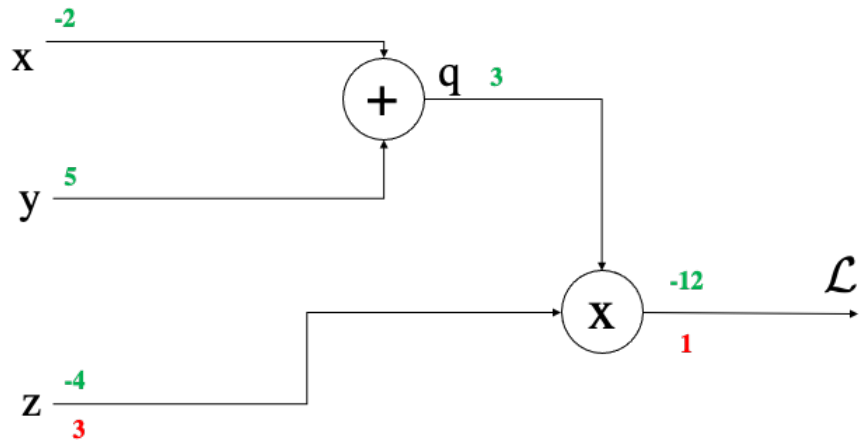
This is what we wanted:

$$\frac{\partial \mathrm{L}}{\partial x}, \frac{\partial \mathrm{L}}{\partial y}, \frac{\partial \mathrm{L}}{\partial z}$$

This is what we have

$$\frac{\partial \mathrm{L}}{\partial q} = z, \frac{\partial \mathrm{L}}{\partial z} = q, \frac{\partial \mathrm{q}}{\partial x} = 1, \frac{\partial \mathrm{q}}{\partial y} = 1$$

Backward Pass



What is the value of the gradient of L on q?:

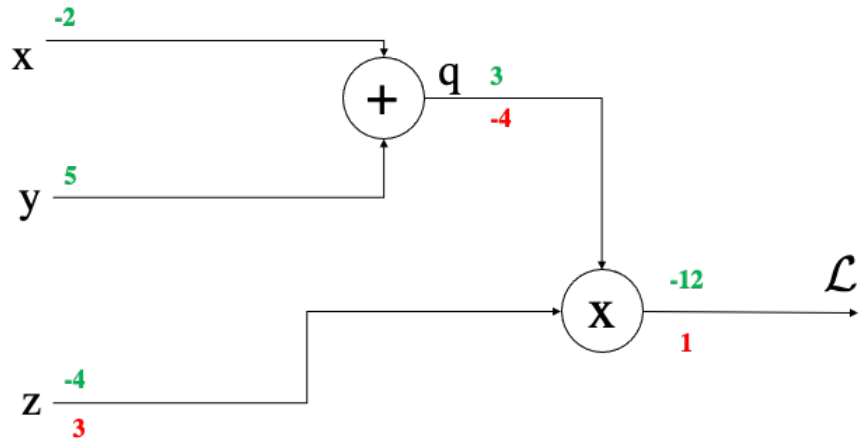
This is what we wanted:

$$\frac{\partial \mathrm{L}}{\partial x}, \frac{\partial \mathrm{L}}{\partial y}, \frac{\partial \mathrm{L}}{\partial z}$$

This is what we have

$$\frac{\partial \mathrm{L}}{\partial q} = z, \frac{\partial \mathrm{L}}{\partial z} = q, \frac{\partial \mathrm{q}}{\partial x} = 1, \frac{\partial \mathrm{q}}{\partial y} = 1$$

Backward Pass



What is the value of the gradient of L on q?:

$$\frac{\partial \mathrm{L}}{\partial q} = z = -4$$

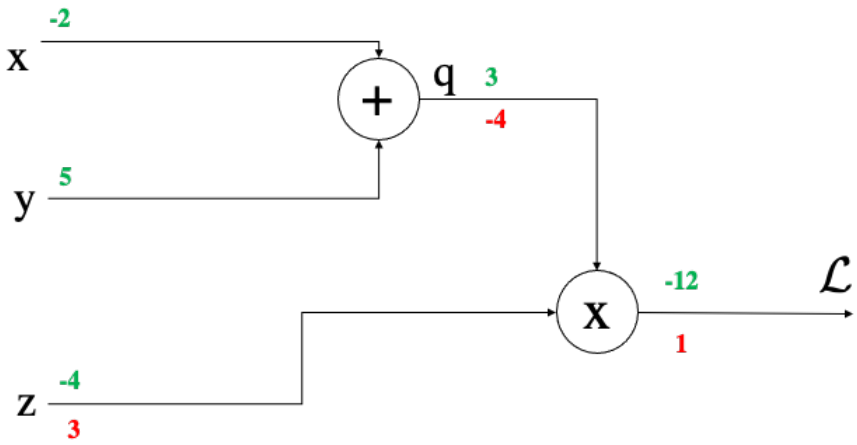
This is what we wanted:

$$\frac{\partial \mathrm{L}}{\partial x}, \frac{\partial \mathrm{L}}{\partial y}, \frac{\partial \mathrm{L}}{\partial z}$$

This is what we have

$$\frac{\partial \mathrm{L}}{\partial q} = z, \frac{\partial \mathrm{L}}{\partial z} = q, \frac{\partial \mathrm{q}}{\partial x} = 1, \frac{\partial \mathrm{q}}{\partial y} = 1$$

Backward Pass



What is the value of the gradient of L on y?:

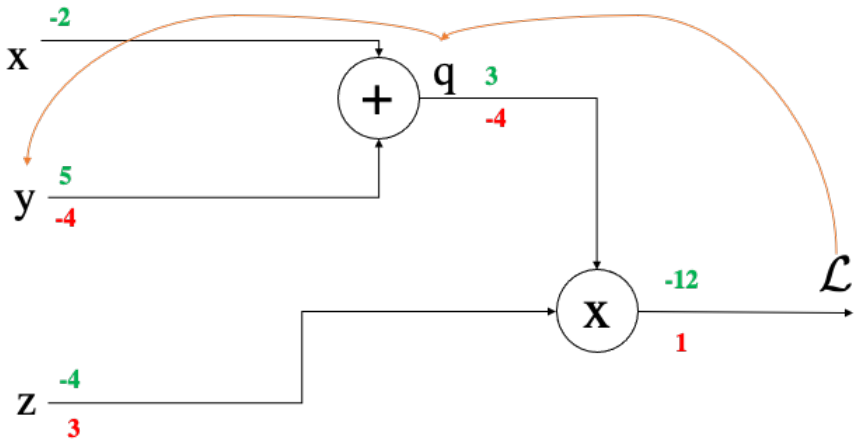
This is what we wanted:

$$\frac{\partial \mathrm{L}}{\partial x}, \frac{\partial \mathrm{L}}{\partial y}, \frac{\partial \mathrm{L}}{\partial z}$$

This is what we have

$$\frac{\partial \mathrm{L}}{\partial q} = z, \frac{\partial \mathrm{L}}{\partial z} = q, \frac{\partial \mathrm{q}}{\partial x} = 1, \frac{\partial \mathrm{q}}{\partial y} = 1$$

Backward Pass



What is the value of the gradient of L on y?:

$$\frac{\partial \mathrm{L}}{\partial y} = \frac{\partial \mathrm{L}}{\partial q}\frac{\partial q}{\partial y} = z \cdot 1 = -4$$

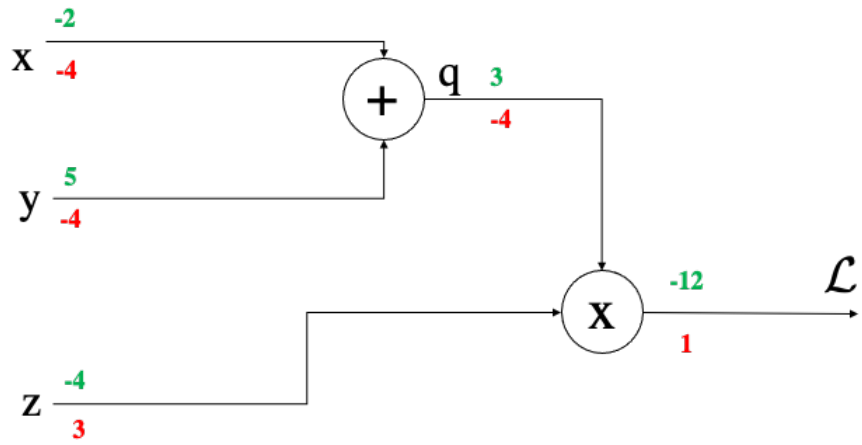
This is what we wanted:

$$\frac{\partial \mathrm{L}}{\partial x}, \frac{\partial \mathrm{L}}{\partial y}, \frac{\partial \mathrm{L}}{\partial z}$$

This is what we have

$$\frac{\partial \mathrm{L}}{\partial q} = z, \frac{\partial \mathrm{L}}{\partial z} = q, \frac{\partial \mathrm{q}}{\partial x} = 1, \frac{\partial \mathrm{q}}{\partial y} = 1$$

Backward Pass



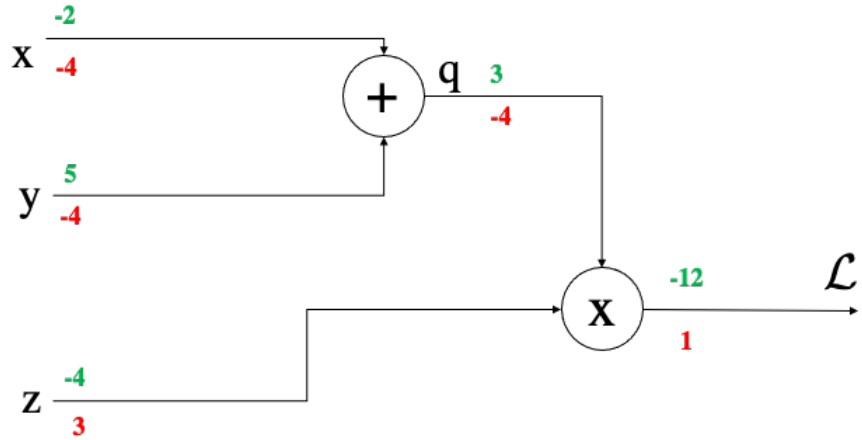
This is what we wanted:

$$\frac{\partial L}{\partial x}, \frac{\partial L}{\partial y}, \frac{\partial L}{\partial z}$$

This is what we have

$$\frac{\partial L}{\partial q} = z, \frac{\partial L}{\partial z} = q, \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

Check with our manual derivation ✅



The high school way (as we did until now):

$$\frac{\partial L(x,y,z)}{\partial x} = (\mathbf{x}z + yz)' = (\mathbf{x}z)' + (yz)' = z = -4$$

$$\frac{\partial L(x,y,z)}{\partial y} = (xz + \mathbf{y}z)' = (xz)' + (\mathbf{y}z)' = z = -4$$

$$\frac{\partial L(x,y,z)}{\partial z} = x + y = +3$$

You know what? I do not trust math, I want to verify with a machine ✅

Pytorch check

```
from torch import tensor

def neural_net(x,y,z):
    return (x+y)*z

x, y, z = tensor(-2., requires_grad=True), tensor(5.,requires_grad=True), tensor(-4.,
requires_grad=True)
loss = neural_net(x,y,z) # forward pass
loss.backward()          # backward (after this I can check the gradients)
for el in [x,y,z]:
    print(el.grad)
```

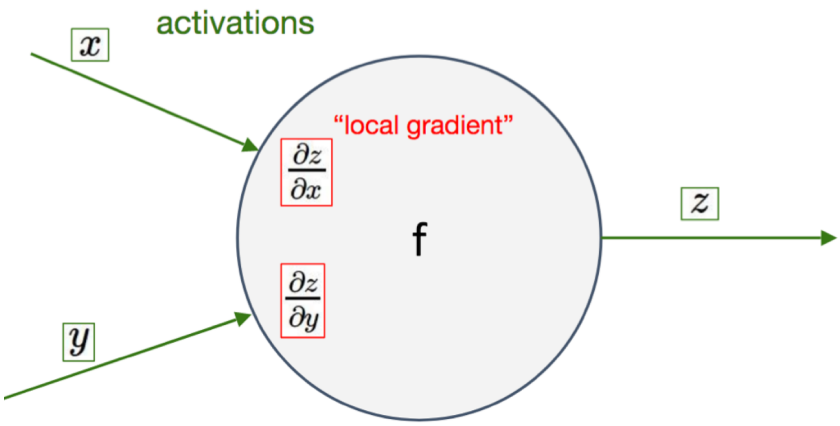
```
tensor(-4.)
tensor(-4.)
tensor(3.)
```

In [17]:
```
from torch import tensor

def neural_net(x,y,z):
    loss = (x+y)*z
    return loss

x, y, z = tensor(-2., requires_grad=True), tensor(5.,requires_grad=True), tensor(-4., requires_grad=True)
loss = neural_net(x,y,z) # forward pass

loss.backward() #backward (ok now I can check the gradients)
for el in [x,y,z]:
    print(el.grad)
```

```
tensor(-4.)
tensor(-4.)
tensor(3.)
```

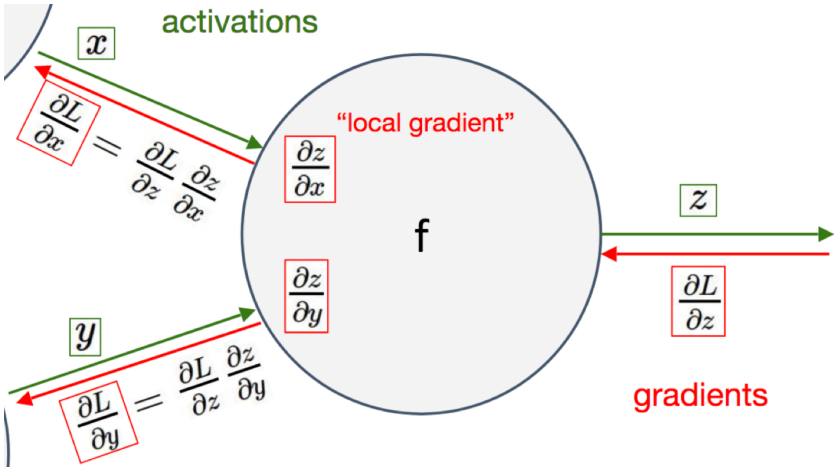
General Recipe for Chain Rule over DAGs [Forward]

Just remember that you have to do at a generic gate:



General Recipe for Chain Rule over DAGs [Backward]

Multiply the gradient that you receive with your local gradient

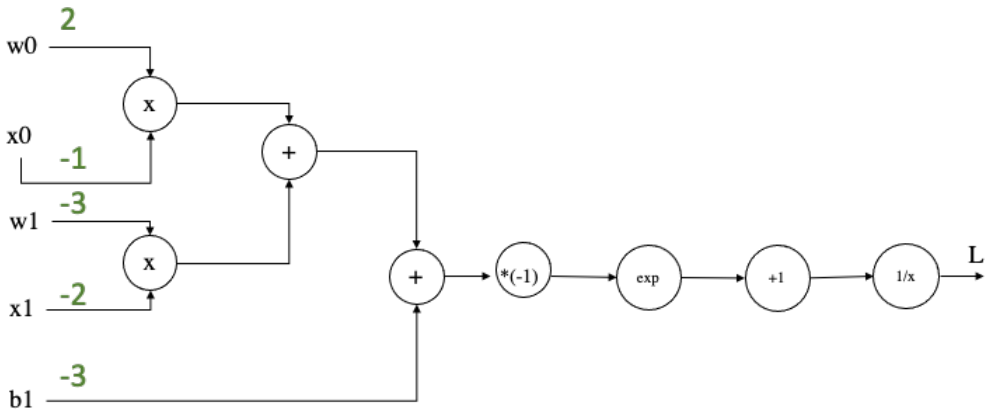


Exam Question lookalike

$$f(w,x) = \frac{1}{1 + e^{-\left(w_0 x_0 + w_1 x_1 + b\right)}}$$

Given the above function, can you perform **forward and backward pass** by writing all the local values and local gradient, by applying the chain rule?

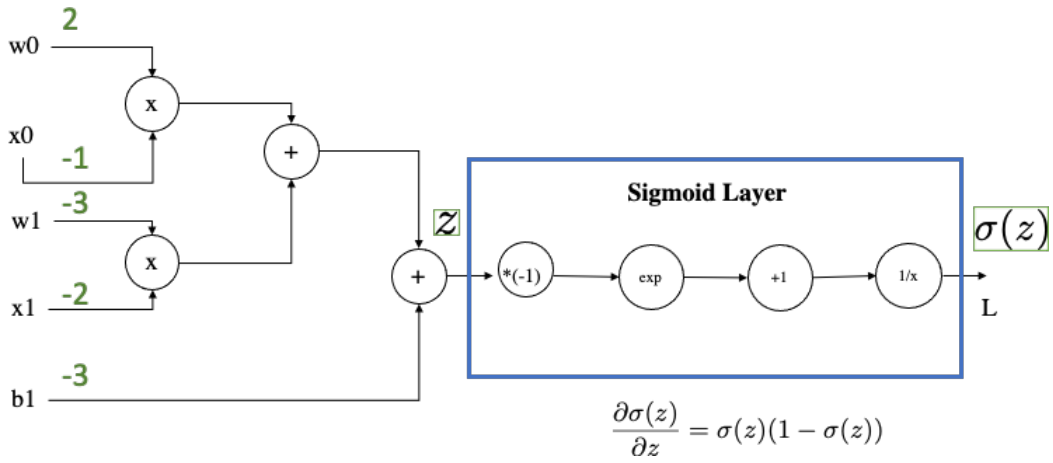
(Direct computation of the gradient does not count for solving it, though you may be using it to double check)



Logistic Regression Computational Graph could be simplified

$$f(w,x) = \frac{1}{1 + e^{-\left(w_0 x_0 + w_1 x_1 + b\right)}}$$

This is what implements the **Sigmoid Layer in Pytorch:**



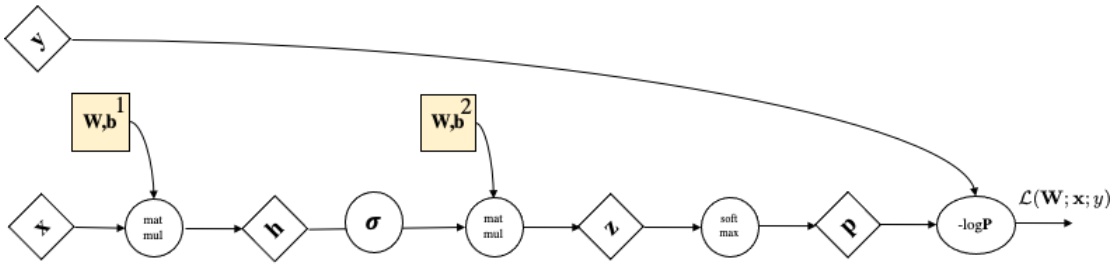
Are we done with training neural nets?

Not completely: till now scalars, but we have matrices and vectors!

Now this looks more familiar

$\forall l \in [1 \ldots, L]$:

1. $\mathbf{W}^l \leftarrow \mathbf{W}^l - \gamma \nabla_{\mathbf{W}^l} \mathrm{L}(\mathbf{x}, y; \{\mathbf{W}, b\})$
2. $\mathbf{b}^l \leftarrow \mathbf{b}^l - \gamma \nabla_{\mathbf{b}^l} \mathrm{L}(\mathbf{x}, y; \{\mathbf{W}, b\})$



Appendix: Gradient of Softmax wrt its probability

Gradient of SoftMax + CE Loss wrt z

$$\begin{aligned}
\mathrm{L}(\mathbf{y}, \hat{\mathbf{y}}) &= -\sum_{j=1}^{q} y_j \log \frac{\exp(z_j)}{\sum_{k=1}^{q} \exp(z_k)} \\
&= \sum_{j=1}^{q} y_j \log \sum_{k=1}^{q} \exp(z_k) - \sum_{j=1}^{q} y_j z_j \\
&= \log \sum_{k=1}^{q} \exp(z_k) - \sum_{j=1}^{q} y_j z_j.
\end{aligned}$$

Gradient of SoftMax + CE Loss wrt z

Below we take the **partial derivative wrt class j, so it is not in vector notation:**

$$\partial_{z_j} \mathrm{L}(\mathbf{y}, \hat{\mathbf{y}}) = \frac{\exp(z_j)}{\sum_{k=1}^{q} \exp(z_k)} - y_j = \mathrm{softmax}(\mathbf{z})_j - y_j$$

It is a GLM!

Gradient of SoftMax + CE Loss wrt z

Below we take the **partial derivative wrt class j, so it is not in vector notation:**

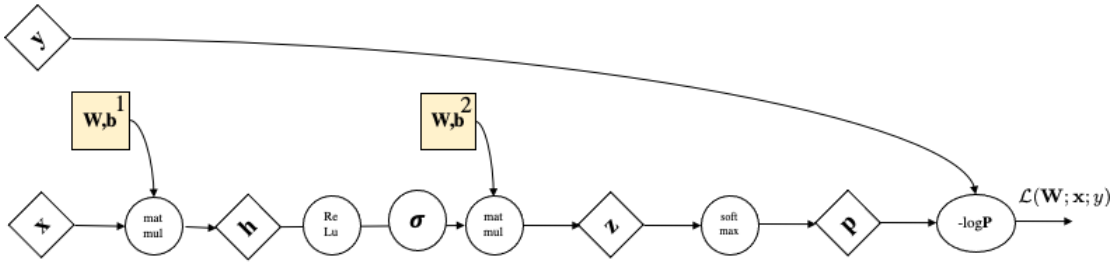
$$\partial_{z_j} \mathrm{L}(\mathbf{y}, \hat{\mathbf{y}}) = \frac{\exp(z_j)}{\sum_{k=1}^{q} \exp(z_k)} - y_j = \mathrm{softmax}(\mathbf{z})_j - y_j$$

```
p = [0.2 0.7 0.1] y =[0 1 0]

dL/dz  = [0.2 0.7-1 0.1] = [0.2 -0.3 0.1]
```

Now this looks more familiar

$\forall l \in [1 \ldots, L]$:

1. $\mathbf{W}^l \leftarrow \mathbf{W}^l - \gamma \nabla_{\mathbf{W}^l} \mathrm{L}(\mathbf{x}, y; \{\mathbf{W}, b\})$
2. $\mathbf{b}^l \leftarrow \mathbf{b}^l - \gamma \nabla_{\mathbf{b}^l} \mathrm{L}(\mathbf{x}, y; \{\mathbf{W}, b\})$



$$\frac{\partial \mathrm{L}}{\partial \mathbf{W}^1} = \frac{\partial \mathrm{L}}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial \mathbf{W}^1}$$

$$\frac{\partial \mathrm{L}}{\partial \mathbf{W}^1} = \frac{\partial \mathrm{L}}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial \sigma} \underbrace{\frac{\partial \sigma}{\partial \mathbf{W}^1}}_{\frac{\partial \mathbf{z}}{\partial \mathbf{W}^1}}$$

$$\frac{\partial \mathrm{L}}{\partial \mathbf{W}^1} = \frac{\partial \mathrm{L}}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial \sigma} \frac{\partial \sigma}{\partial \mathbf{h}} \underbrace{\frac{\partial \mathbf{h}}{\partial \mathbf{W}^1}}_{\frac{\partial \sigma}{\partial \mathbf{W}^1}}$$