Artificial Intelligence and Machine Learning

Unit II

Backpropagation (Intro to Deep Learning)

My own latex definitions

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_24498/1496334134.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 [3]:
```
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_24498/1496334134.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**
- Optimization in Deep Learing with **SGD over mini-batch with momentum**
- MLP and Fully-Connected Neural Nets
- Intro to Backpropagation

Today's lecture

Supervised, Parametric Models

Propaedeutic part for Deep Learning

0) Go back to Backpropagation

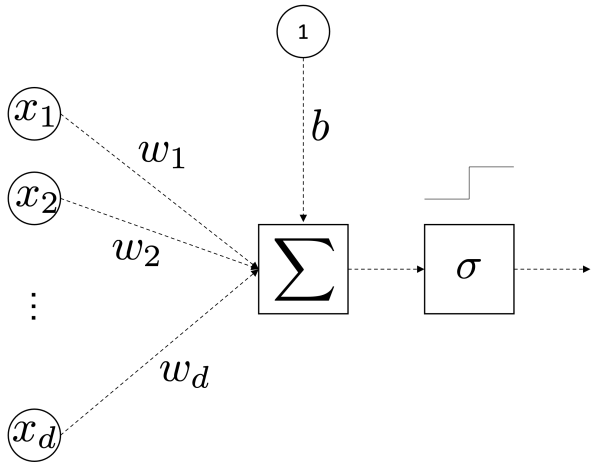
1) Backpropagation with matrices and vectors (Jacobians, Gradients)

2) End of the course! 🤖 🤯

This lecture material is taken from

- d2l.ai - Multi Variable Calculus
- Karpathy (Tesla Machine Learning Director) 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

Now you see why it's named Multi-Layer Perceptron (MLP)



Representation of a Single Layer

Let's consider our linear softmax regressor

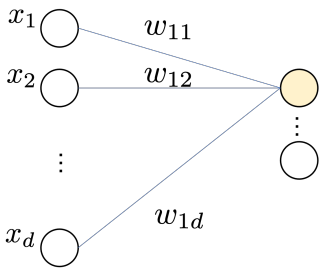
$$\underbrace{\mathbf{z}}_{\mathbb{R}^{Kx1}} = \underbrace{\mathbf{W}}_{\mathbb{R}^{K\times d}}\underbrace{\mathbf{x}}_{\mathbb{R}^{d\times 1}} + \underbrace{\mathbf{b}}_{\mathbb{R}^{K}}$$

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

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



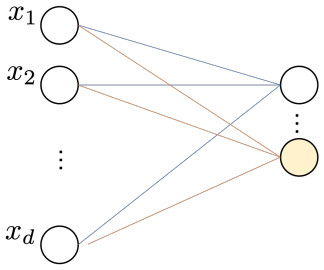
$$\mathbf{W}_1^T\mathbf{x} + b_1$$

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

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



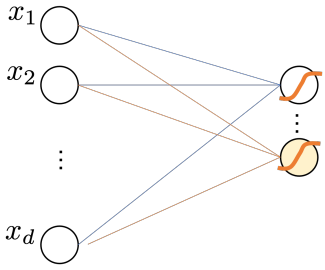
$$\mathbf{W}_k^T\mathbf{x} + b_k$$

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

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



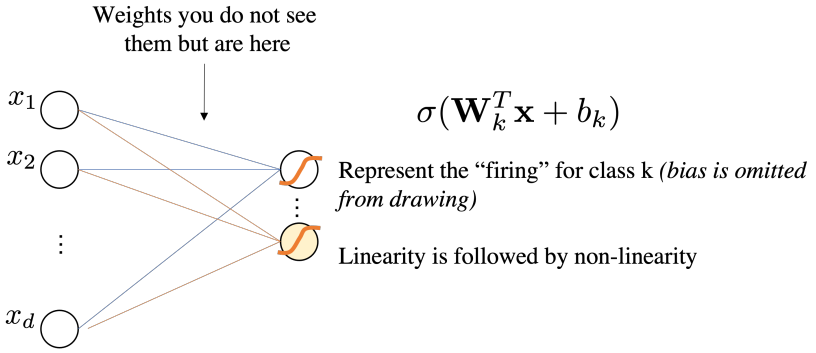
$$\sigma(\mathbf{W}_k^T\mathbf{x} + b_k)$$

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

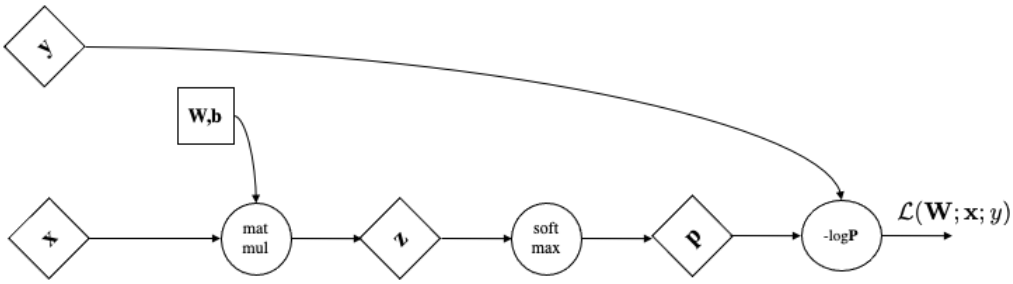
Linearity is followed by non-linearity

Representation of a Single Layer: Linear plus non-Linear

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



Representation as a computational graph



Adding another non-linear layer before the classifier

- We improve the *expressiveness* of our learned function by adding another **NON-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 $\phi(\mathbf{x})$.
- Sorry, notation becomes complex. Upper script means layer index; lower-script selects the unit
- $\mathbf{W}^1 \in \mathbb{R}^{d\times p}$, $\boldsymbol{b}^1 \in \mathbb{R}^p$ so then $\mathbf{W}^2 \in \mathbb{R}^{p\times k}$, $\boldsymbol{b}^2 \in \mathbb{R}^k$

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

$$\text{dim. analysis:} \quad d \mapsto p \mapsto k$$

$\mathbf{W}^1 \in \mathbb{R}^{d\times p}$ 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.

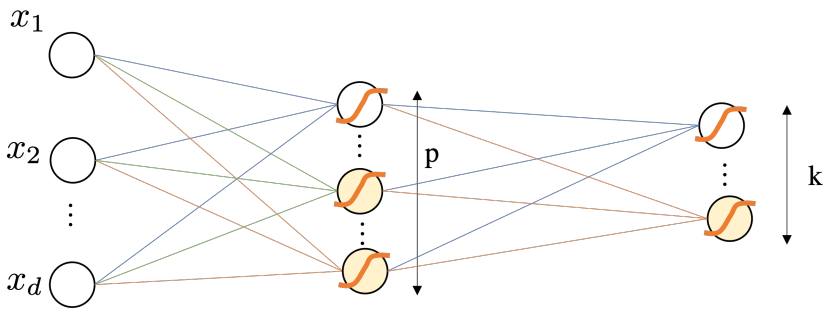
$$\mathbf{p} = \sigma(\mathbf{W}^2 \underbrace{\Big(\sigma(\mathbf{W}^1\mathbf{x} + \boldsymbol{b}^1)\Big)}_{\phi(x)} + \boldsymbol{b}^2)$$

$$\text{dim. analysis:} \quad d \mapsto p \mapsto k$$

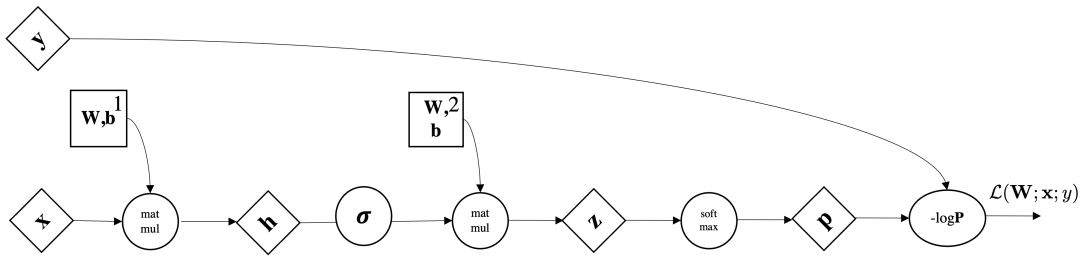
Let's update our visualizations

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: ReLu - Rectified Linear Unit

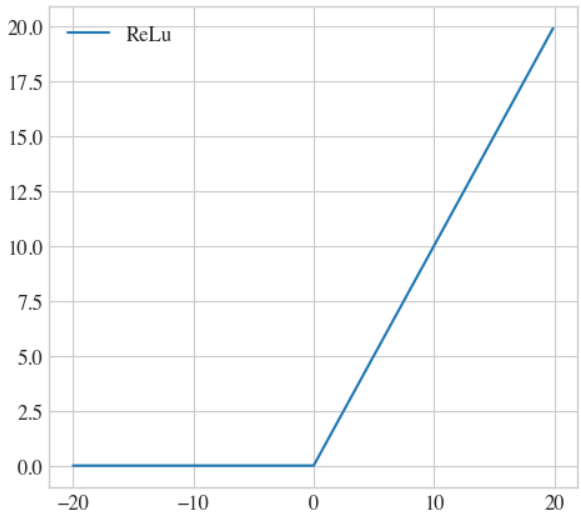
Very important: **Activation Functions are computed element-wise.**

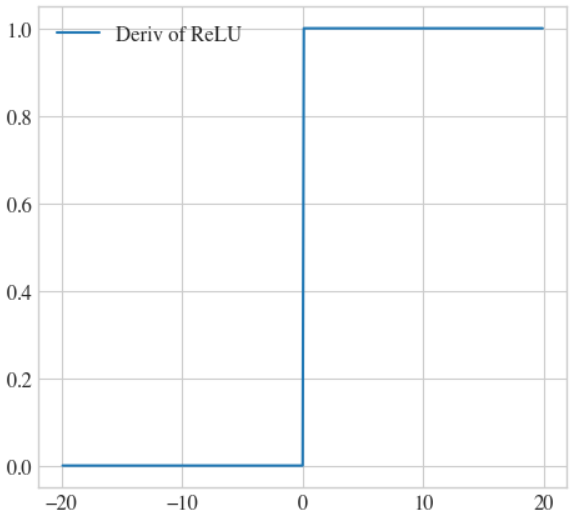
$$\sigma(z) = \max(0, z) \quad \text{ReLu}$$

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 [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 = 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

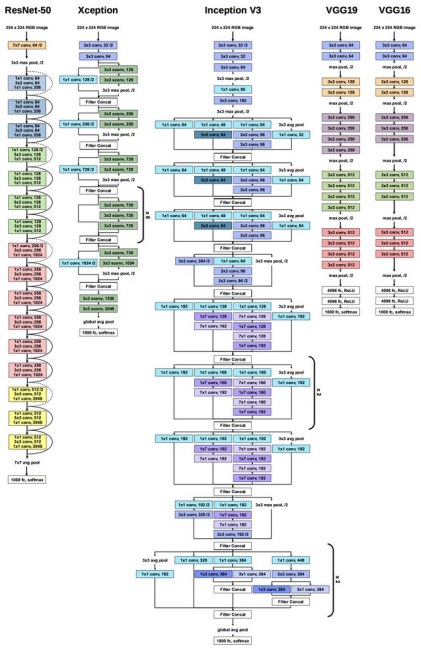
- Used to model output probability
- Nowadays not used in middle layers
- Have to compute $\exp()$
- **Vanishing gradients** for large input magnitude

ReLU

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

Backpropagation and Differential Programming

NN can be huge composition of functions! 😱



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

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)** 💻

1. Infeasible to derive manully the gradient update, let the machines work for us

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

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

- **[Offline] Evaluate your NN loss at current weight value** `L(w)`

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

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

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

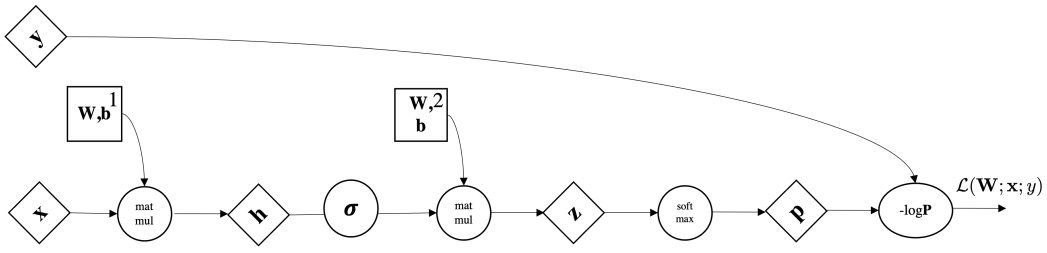
At the end you have your **numerical gradients** $\nabla_{\mathbf{W}}\mathcal{L}_{ij}$.

3. Backpropagation

Let's be clear on what we need to compute

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

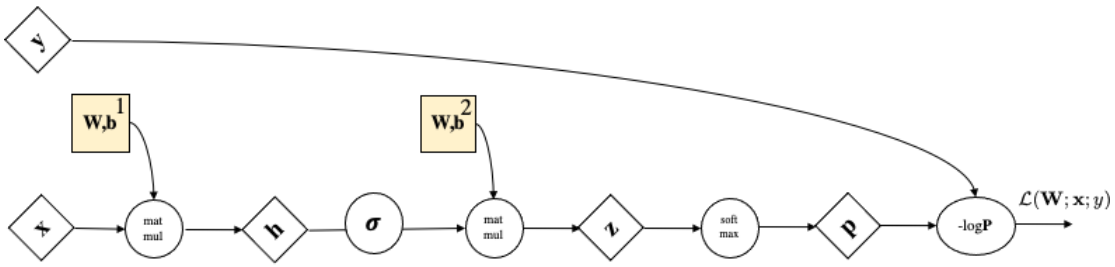
1. $\nabla_{\mathbf{W}^l}\mathcal{L}(\mathbf{x}, y; \{\mathbf{W}, b\})$
2. $\nabla_{\mathbf{b}^l}\mathcal{L}(\mathbf{x}, y; \{\mathbf{W}, b\})$



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

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

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



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**?

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

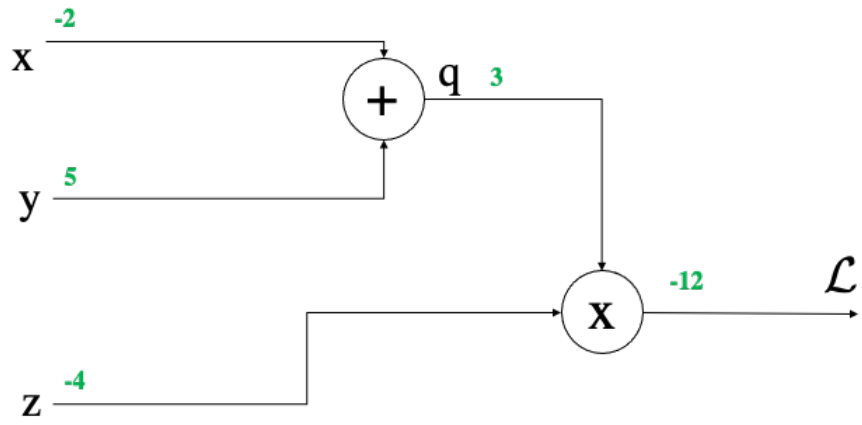
$$\frac{\partial loss}{\partial x} = \frac{\partial loss}{\partial g}\frac{\partial g}{\partial h}\frac{\partial h}{\partial i}\frac{\partial i}{\partial x}$$

Chain Rule on Directed Acyclic Graph (DAG)

Automate the computation of derivatives with computer science

1. Forward Pass
2. Backward Pass

Forward Pass



This is what we wanted:

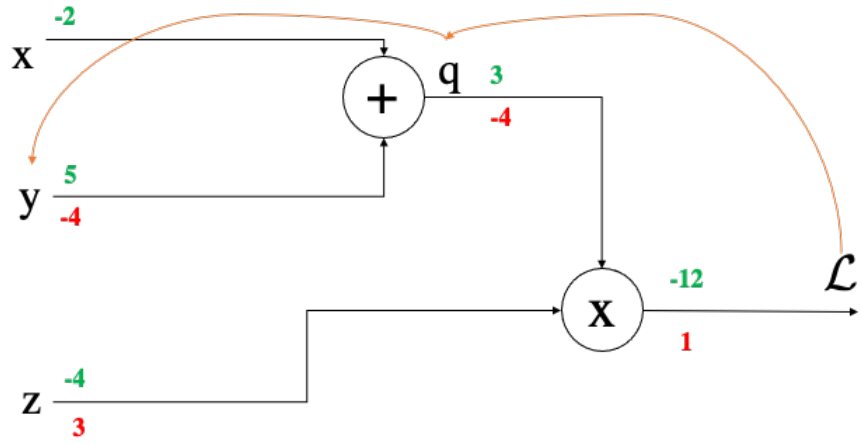
$$\frac{\partial \mathcal{L}}{\partial x}, \frac{\partial \mathcal{L}}{\partial y}, \frac{\partial \mathcal{L}}{\partial z}$$

Forward pass

1. Evaluate the function on input *(the function is "hard-coded" with your model/code)*
2. Store **"local" derivative** at each layer/gate

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

Backward Pass



What is the value of the gradient of $\mathcal{L}$ on y?:

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

This is what we wanted:

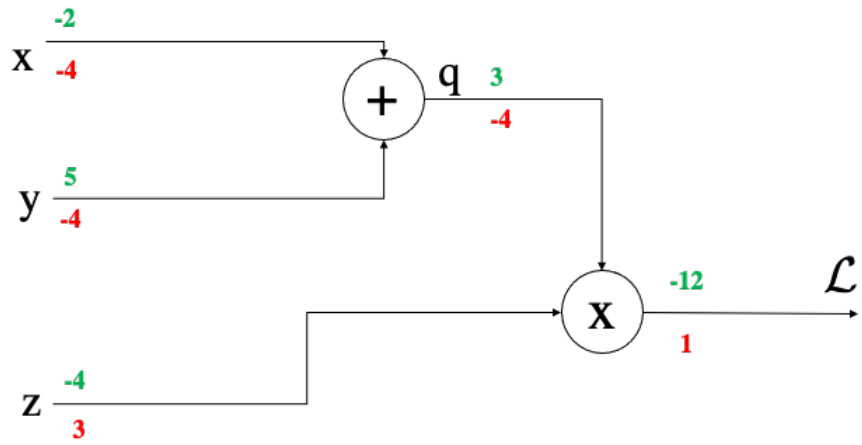
$$\frac{\partial \mathcal{L}}{\partial x}, \frac{\partial \mathcal{L}}{\partial y}, \frac{\partial \mathcal{L}}{\partial z}$$

Backward pass

1. Start from loss scalar value
2. Backpropagate the current derivative/gradient to higher layers
3. Use chain rule to aggregate a) local derivative b) what arrives from "the top"

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

Backward Pass



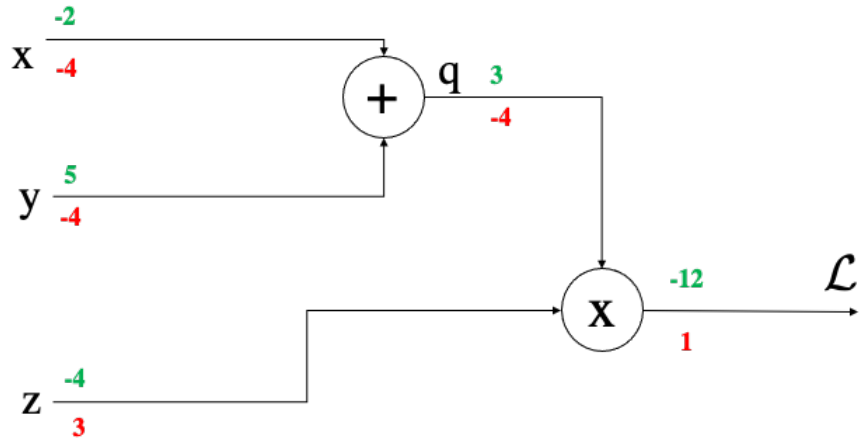
This is what we wanted:

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

This is what we have

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

Check with our manual derivation ✅



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

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

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

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

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.)
```

Pytorch check

- Pytorch creates a **dynamic computational directed acyclic graph (DAG) under the hood.**

- We can also see the graph with Pytorch if you want (`torchviz` simplifies the plot)
- `! pip install torchviz`

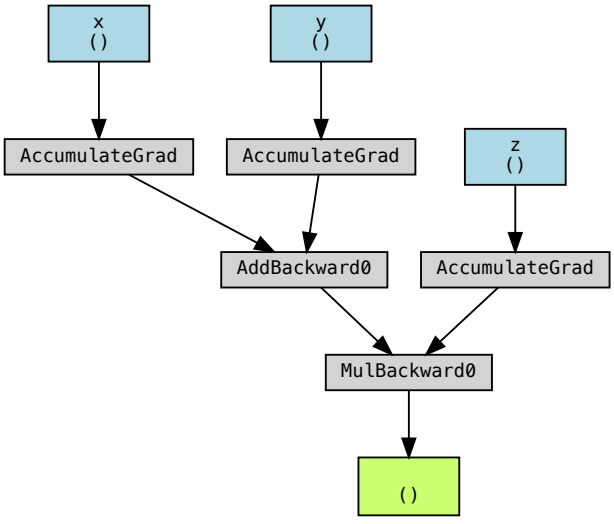
In [5]:
```
from torchviz import make_dot
from torch import tensor
def neural_net(x,y,z):
    return (x+y)*z

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 (ok now I can check the gradients)
for el in [x, y, z]:
    print(el.grad)
print(loss)
make_dot(loss, params=dict([('x', x),('y', y),('z', z)]))
```

```
tensor(-4.)
tensor(-4.)
tensor(3.)
tensor(-12., grad_fn=<MulBackward0>)
```

Out[5]:

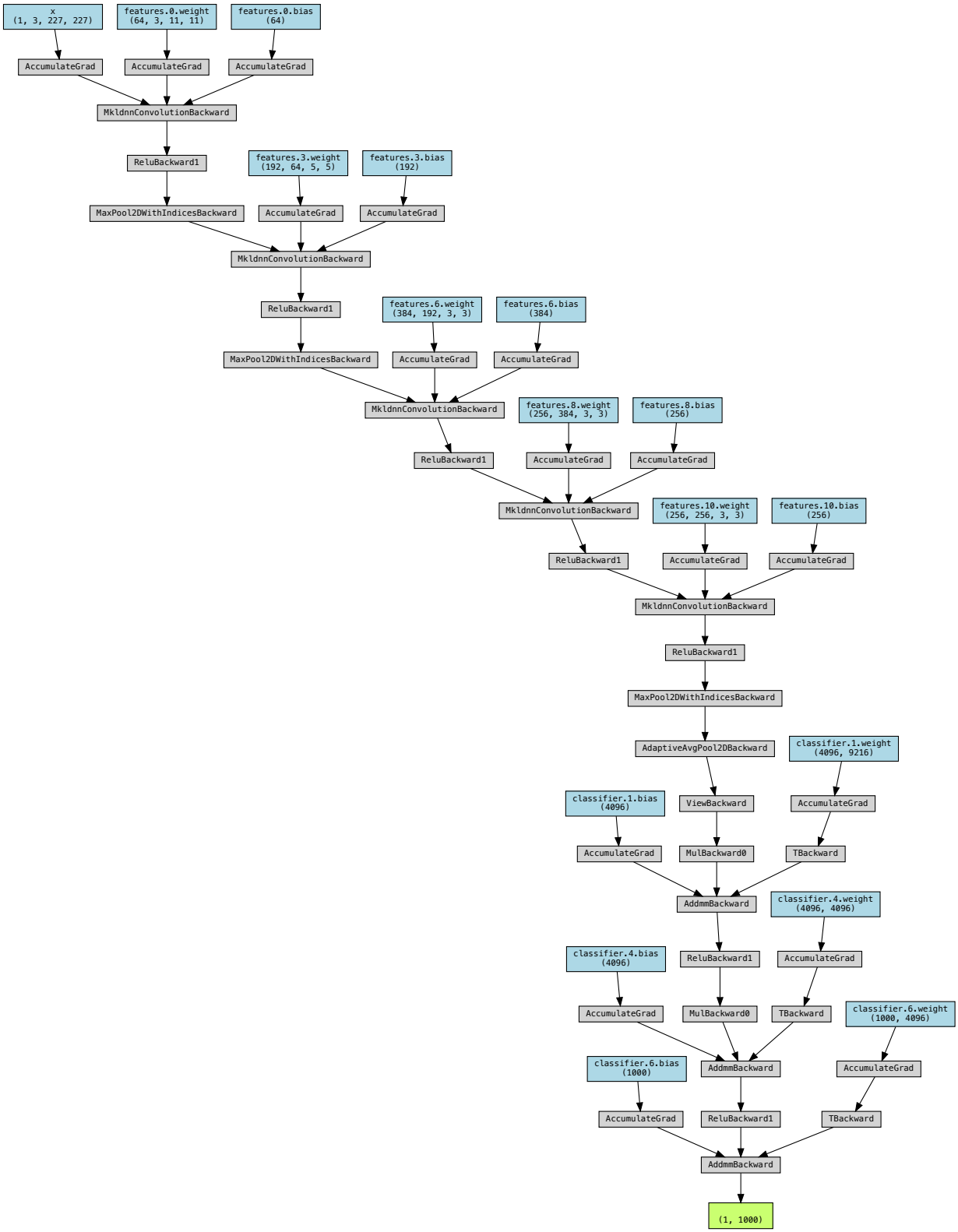


We can also see the DAG of AlexNet... [paper from 2012]

...but it won't fit my screen

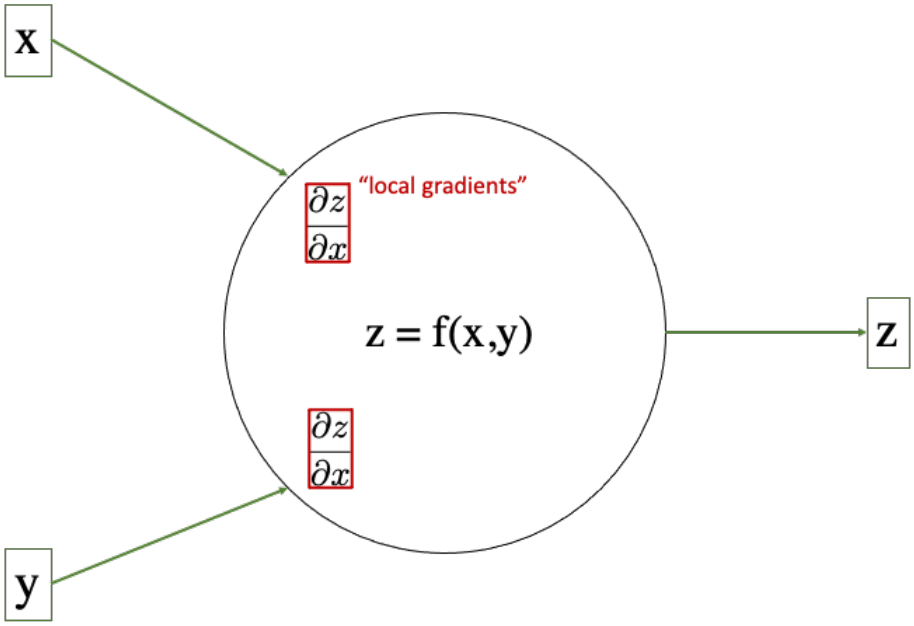
In [6]:
```
import torch
from torchvision.models import AlexNet
model = AlexNet()
x = torch.randn(1, 3, 227, 227).requires_grad_(True)
y = model(x)
make_dot(y, params=dict(list(model.named_parameters()) + [('x', x)]))
```

Out[6]:



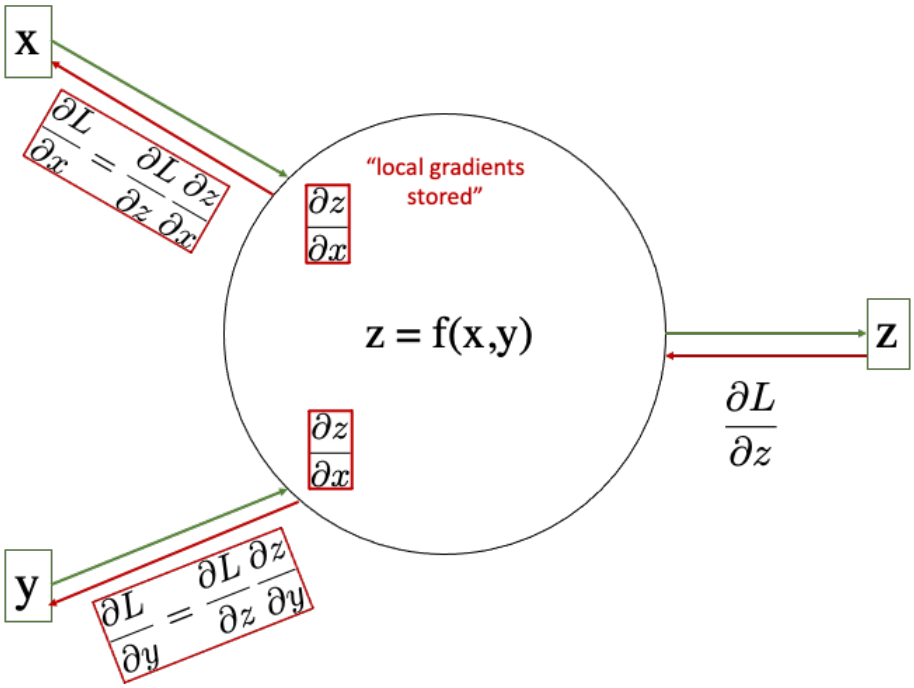
General Recipe for Chain Rule over DAGs [Forward]

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

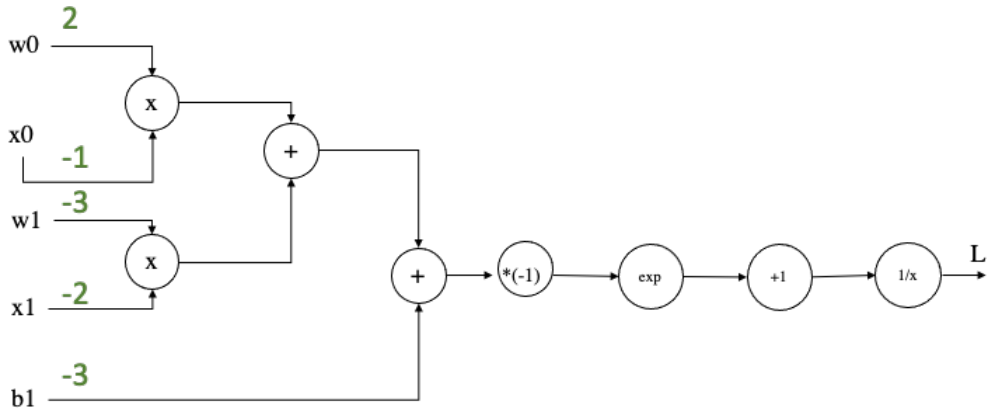


Exam Question Lookalike

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+b)}}$$

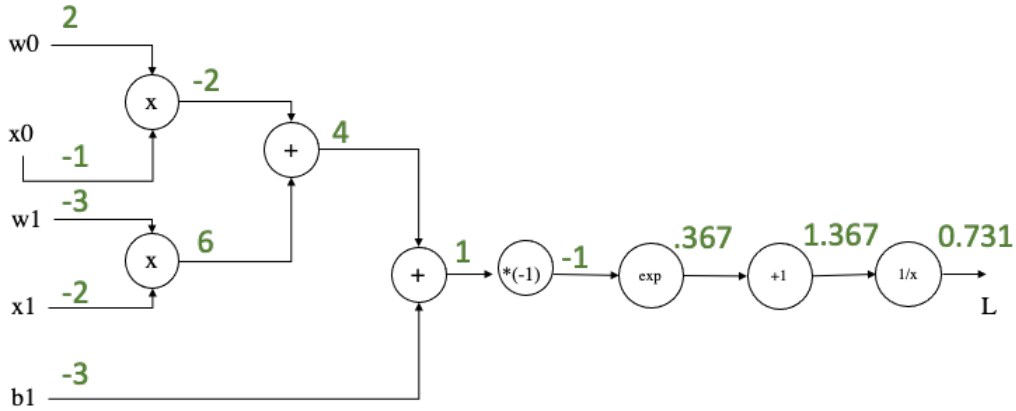
Given the above function, can you perform **forward and backward pass** by writing all the local values and local gradient, by applying the chain rule?

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



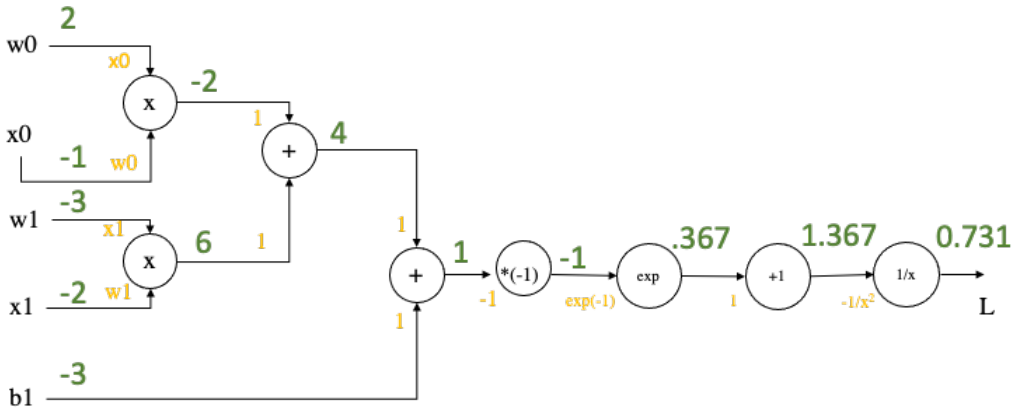
Exam Question Lookalike [Forward Pass]

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+b)}}$$



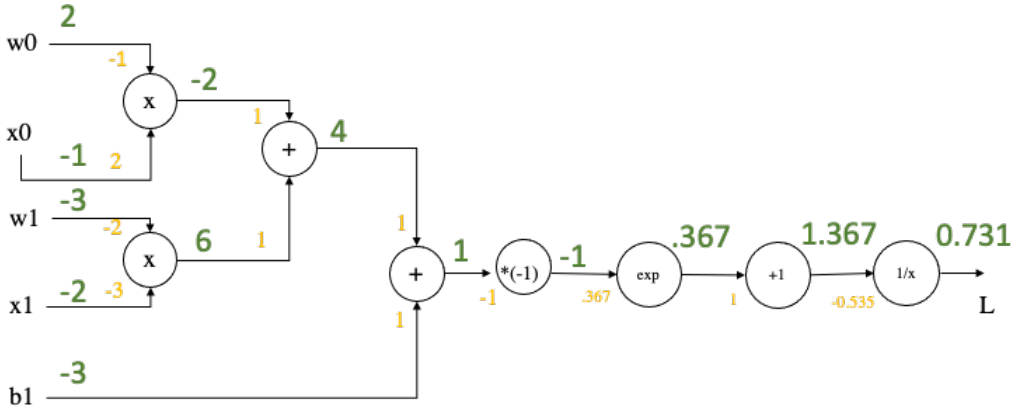
Do not forget to store local gradients!

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+b)}}$$



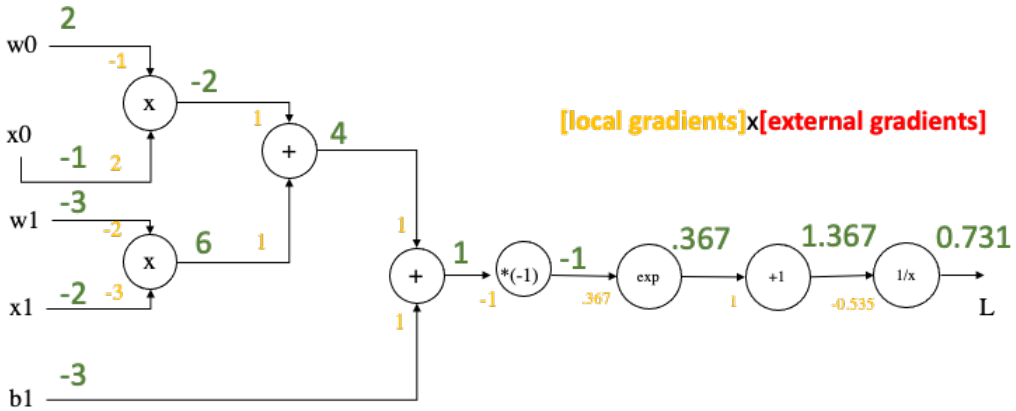
Local gradients evaluated at input values! Now ready to go backwards

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+b)}}$$



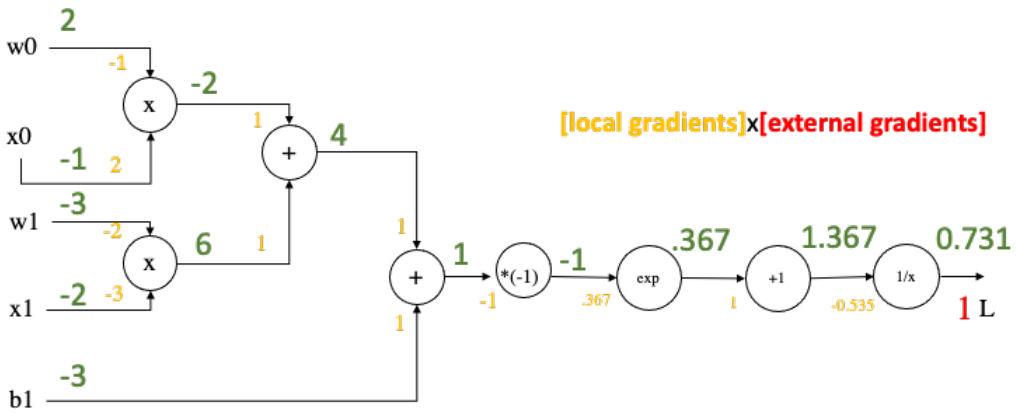
End Forward; Now Ready to Backprop [Backward]

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+b)}}$$



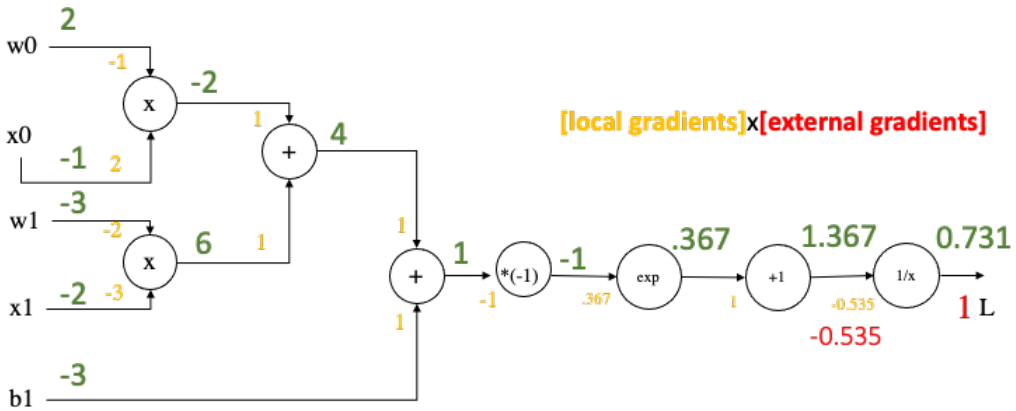
End Forward; Now Ready to Backprop [Backward]

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+b)}}$$



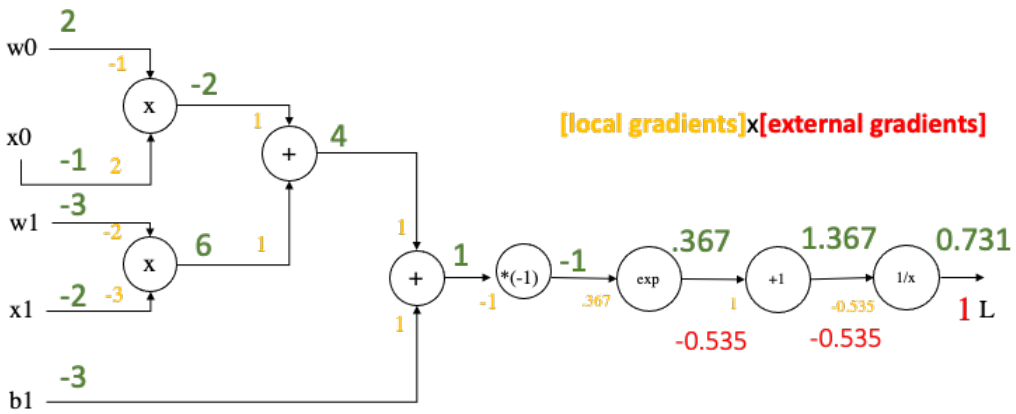
End Forward; Now Ready to Backprop [Backward]

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+b)}}$$



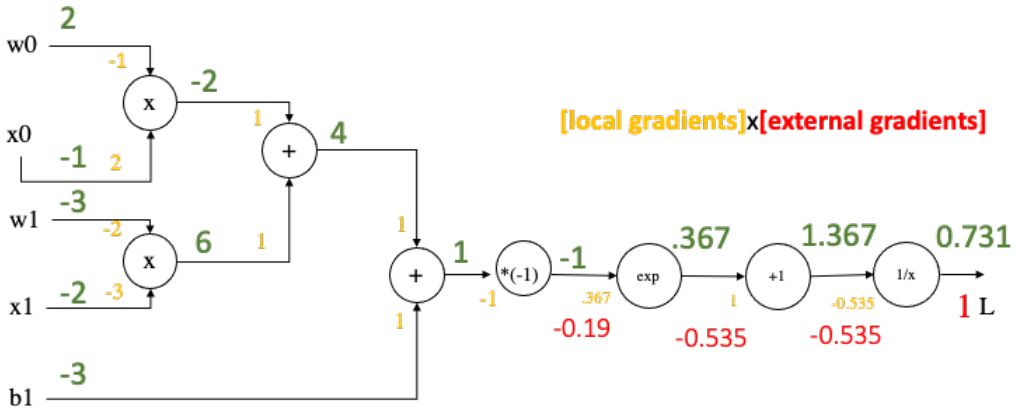
End Forward; Now Ready to Backprop [Backward]

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+b)}}$$



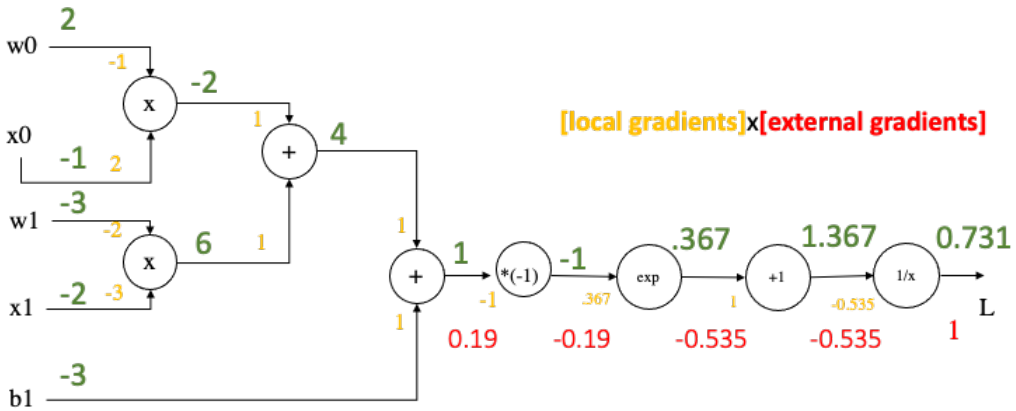
End Forward; Now Ready to Backprop [Backward]

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+b)}}$$



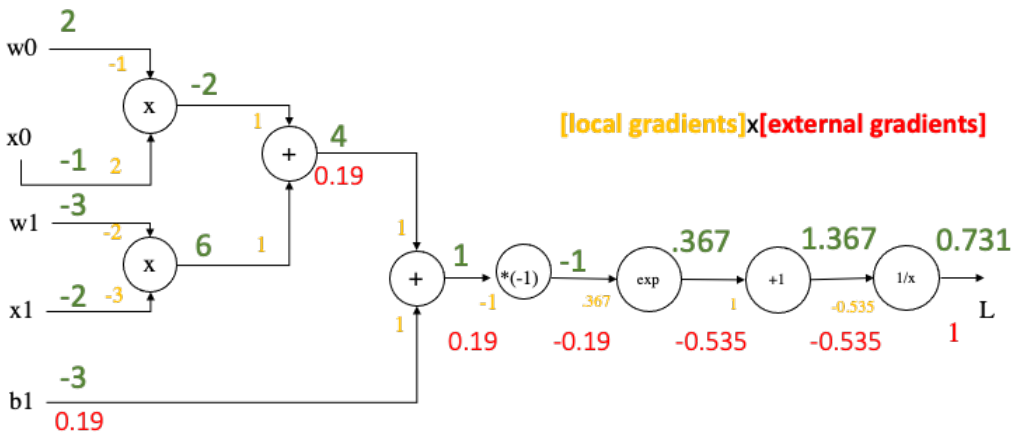
End Forward; Now Ready to Backprop [Backward]

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+b)}}$$



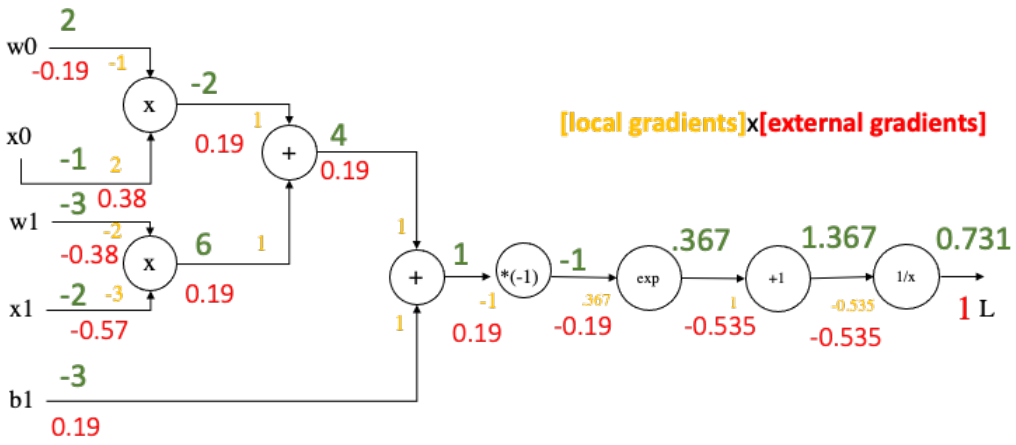
End Forward; Now Ready to Backprop [Backward]

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+b)}}$$



End Backward

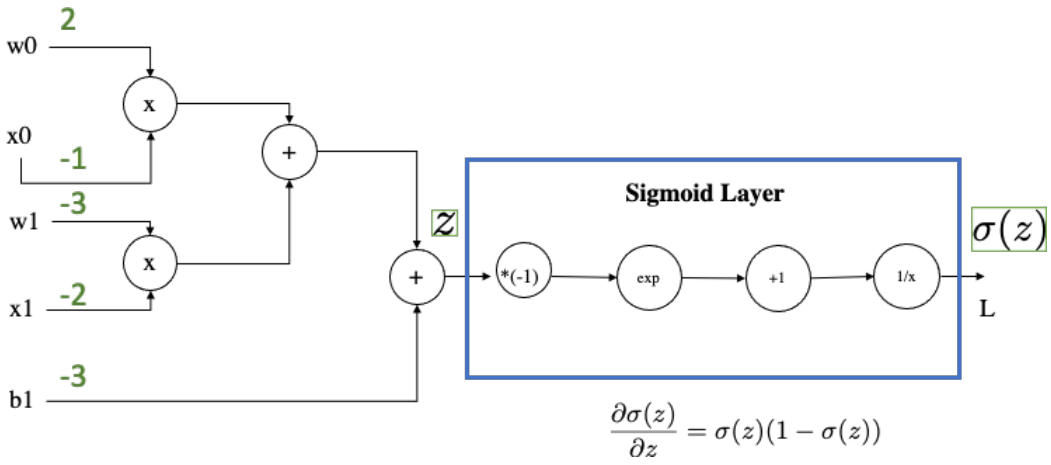
$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+b)}}$$



Logistic Regression Computational Graph could be simplified

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+b)}}$$

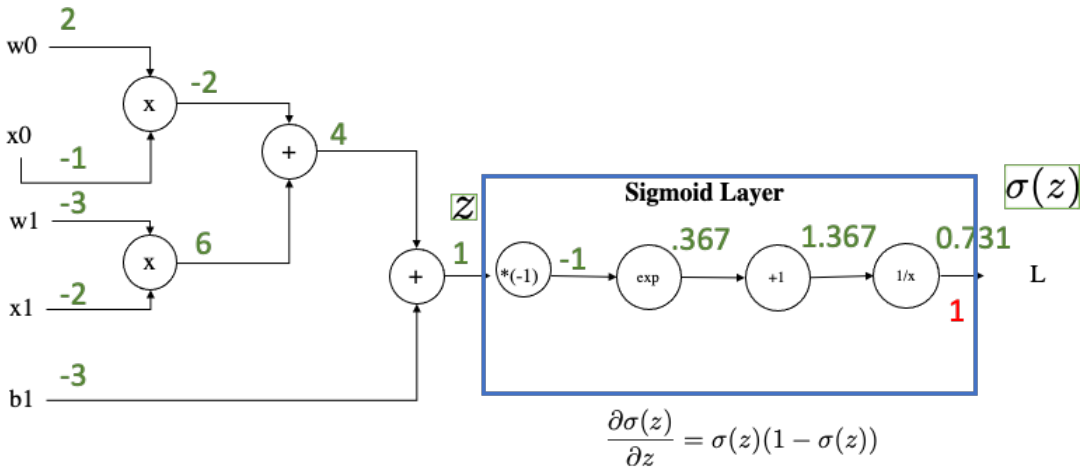
This is what implements the **Sigmoid Layer in Pytorch:**



Logistic Regression Computational Graph could be simplified

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+b)}}$$

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



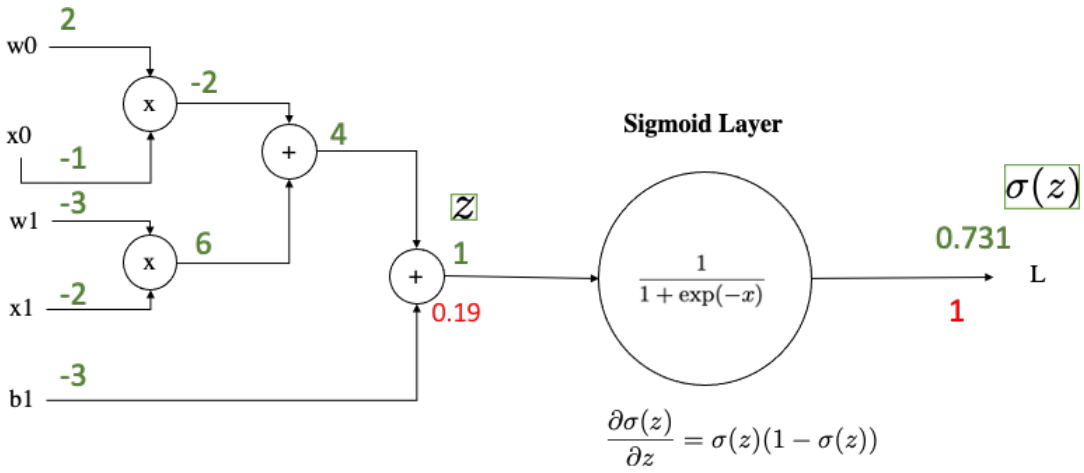
$$\frac{\partial \sigma(z)}{\partial z} = \sigma(z)(1-\sigma(z))$$

1/(1+exp(-1))*(1-1/(1+exp(-1))) ~ 0.19 x 1

Logistic Regression Computational Graph could be simplified

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+b)}}$$

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



$$\frac{\partial \sigma(z)}{\partial z} = \sigma(z)(1-\sigma(z))$$

1/(1+exp(-1))*(1-1/(1+exp(-1))) ~ 0.19 x 1

What PyTorch does very roughly [Pseudo code]

```
class ComputationalGraph:

    def forward(inputs):
        #1. pass inputs to input layers/gate
        #2. forward the computational graph
        for gate in self.graph.nodes_topologically_sorted():
            gate.forward()
        return loss # the finale gate outputs the loss

    def backward():
        for gate in reversed(self.graph.nodes_topologically_sorted()):
            gate.backward() # chain rule applied with local grads
        return inputs_grads
```

What is a Gate (ex. Multiplicative Scalar Gate) [Pseudo code]

```
class MultiplyGate:

    def forward(x,y):
        self.x = x # save local grads dz/dy
        self.y = y # save local grads dz/dx
        return x*y # eval function z = x + y

    def backward(dz):
        dx = self.y * dz # [dz/dx * dL/dz]
        dy = self.x * dz # [dz/dy * dL/dz]
        return [dx, dy]
```

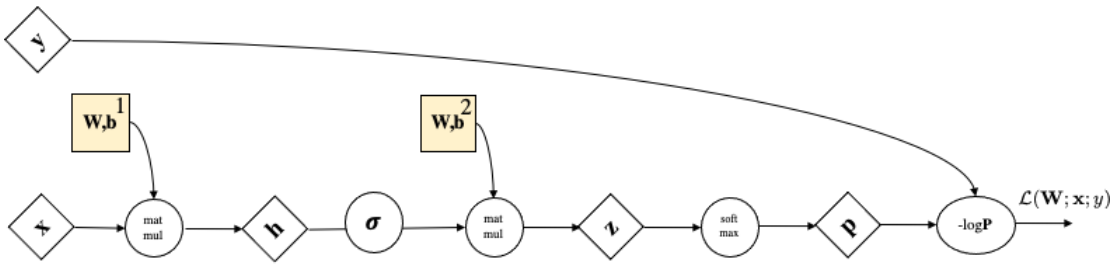
Are we done with training neural nets?

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

Now this looks more familiar

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

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



Appendix: Gradient of Softmax wrt its probability

Gradient of SoftMax + CE Loss wrt z

$$
\begin{aligned}
\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) &= -\sum_{j=1}^{q} y_j \log \frac{\exp(z_j)}{\sum_{k=1}^{q} \exp(z_k)} \\
&= \sum_{j=1}^{q} y_j \underbrace{\log \sum_{k=1}^{q} \exp(z_k)}_{\text{not vary with j = constant}} - \sum_{j=1}^{q} y_j z_j \\
&= \log \sum_{k=1}^{q} \exp(z_k) \underbrace{\sum_{j=1}^{q} y_j}_{=1} - \sum_{j=1}^{q} y_j z_j \\
&= \log \sum_{k=1}^{q} \exp(z_k) - \sum_{j=1}^{q} y_j z_j.
\end{aligned}
$$

Taken from here

Gradient of SoftMax + CE Loss wrt z

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

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

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:**

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

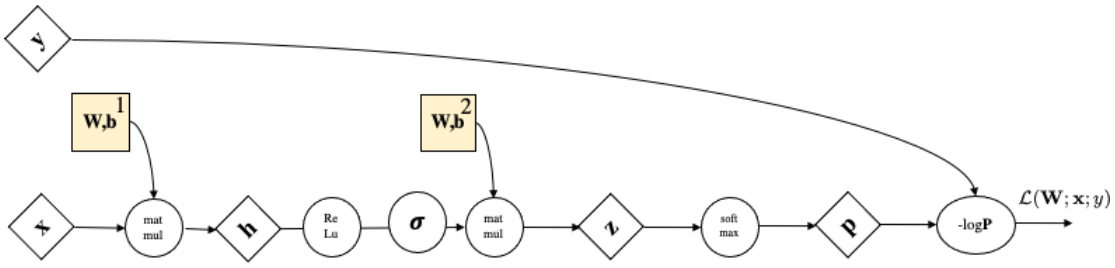
```
p = [0.2 0.7 0.1] y =[0 1 0]

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

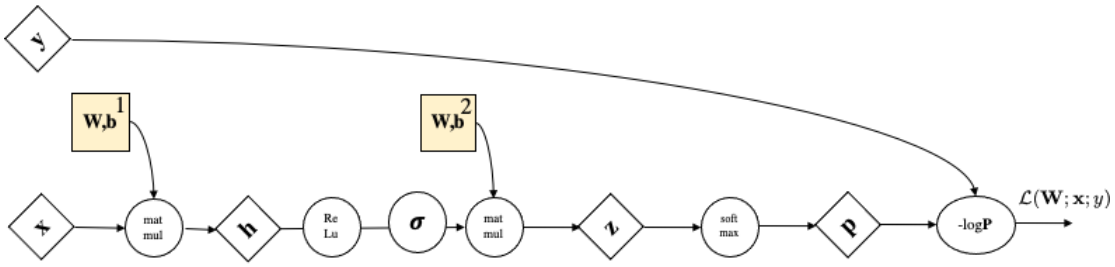


$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^1} = \ ?$$

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

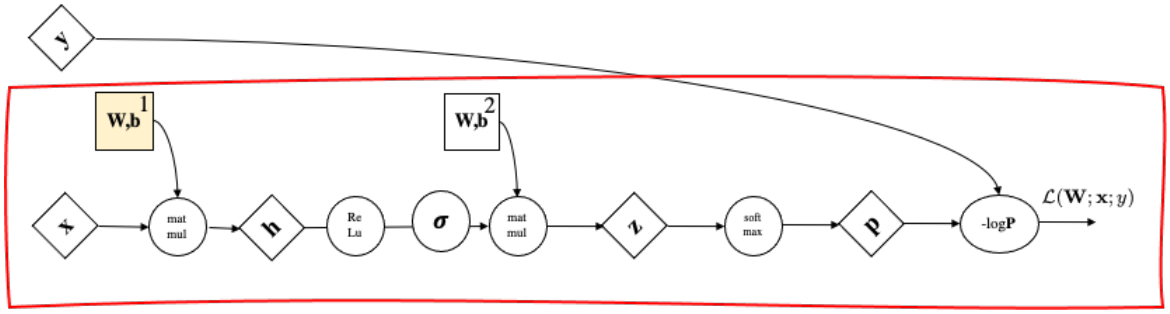
Backprop

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



Backprop

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

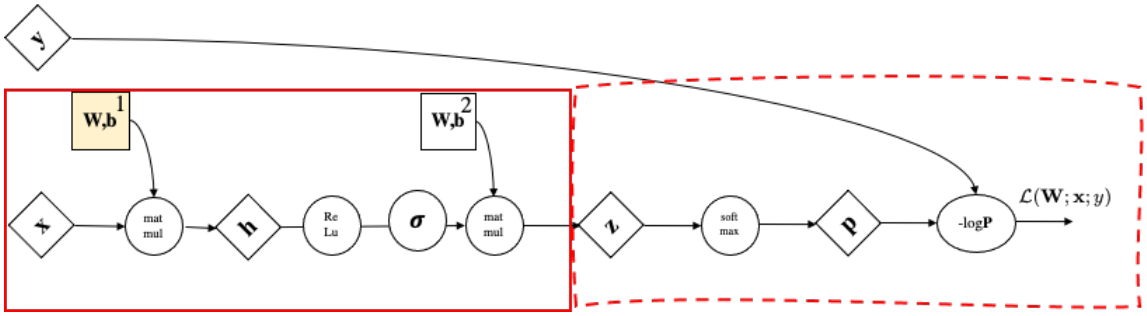


Backprop

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

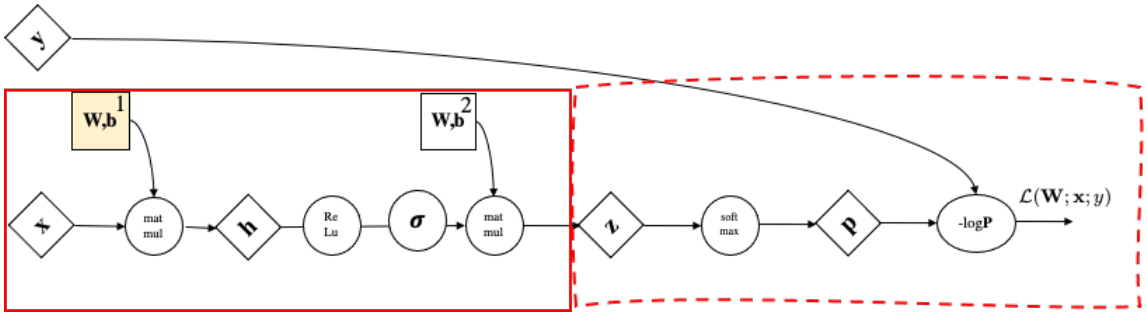
$$\mathcal{L} : \mathbf{z} \mapsto \text{scalar}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}} \in \mathbb{R}^{Z \times 1}$$



Backprop

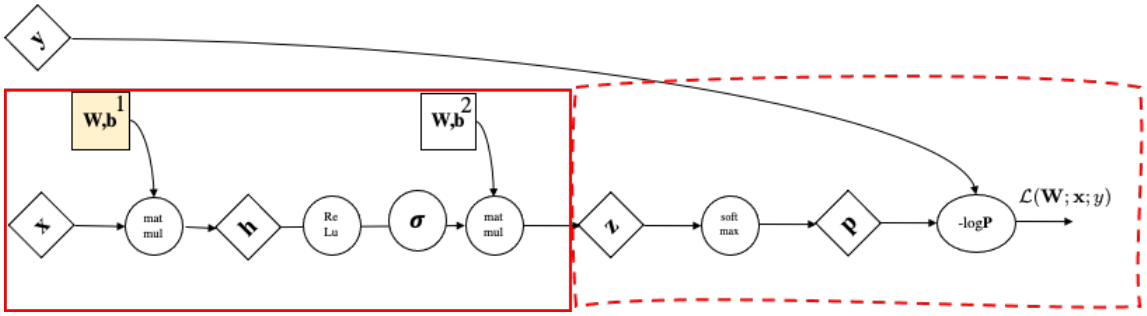
$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^1} = \underbrace{\frac{\partial \mathcal{L}}{\partial \mathbf{z}}}_{\mathbb{R}^{1\times Z}} \underbrace{\frac{\partial \mathbf{z}}{\partial \sigma} \frac{\partial \sigma}{\partial \mathbf{W}^1}}_{\frac{\partial \mathbf{z}}{\partial \mathbf{W}^1}}$$



Backprop

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^1} = \underbrace{\frac{\partial \mathcal{L}}{\partial \mathbf{z}}}_{\mathbb{R}^{1\times Z}} \underbrace{\frac{\partial \mathbf{z}}{\partial \sigma} \frac{\partial \sigma}{\partial \mathbf{W}^1}}_{\frac{\partial \mathbf{z}}{\partial \mathbf{W}^1}}$$

$$\frac{\partial \mathbf{z}}{\partial \sigma} = \ ?$$

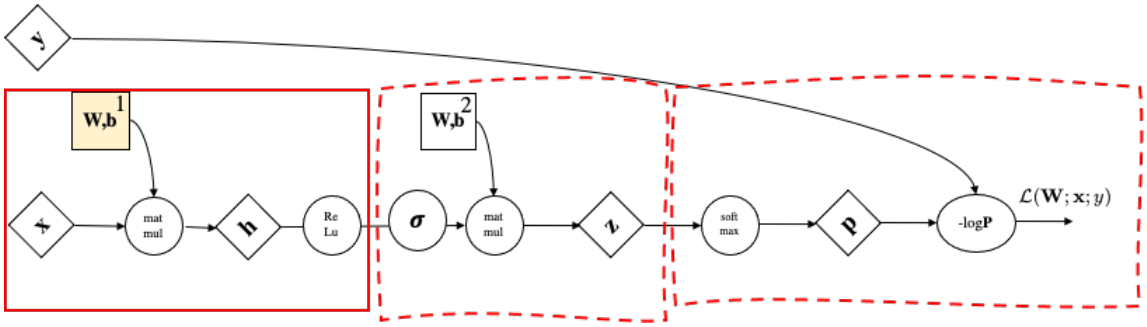


Backprop

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^1} = \underbrace{\frac{\partial \mathcal{L}}{\partial \mathbf{z}}}_{\mathbb{R}^{1\times Z}} \underbrace{\frac{\partial \mathbf{z}}{\partial \sigma} \frac{\partial \sigma}{\partial \mathbf{W}^1}}_{\frac{\partial \mathbf{z}}{\partial \mathbf{W}^1}}$$

$f : \sigma \mapsto \mathbf{z}$ vector to vector

$$\frac{\partial \mathbf{z}}{\partial \sigma} = \ ? \ ; \mathbf{z} = \mathbf{W}\sigma + \mathbf{b}$$

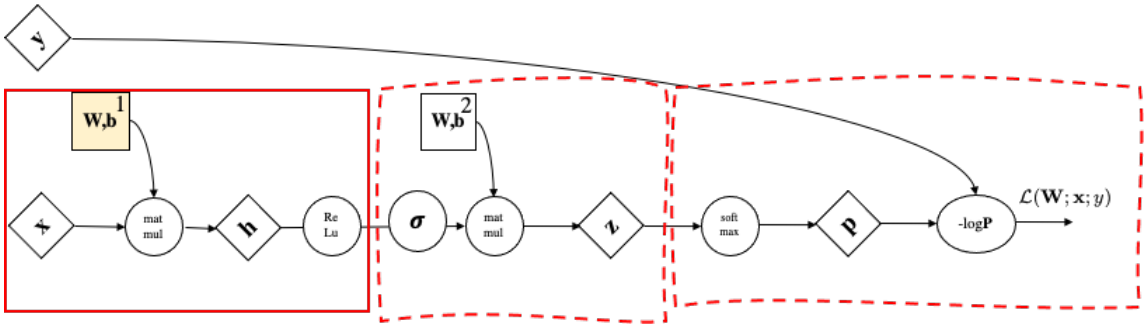


Backprop

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^1} = \underbrace{\frac{\partial \mathcal{L}}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial \sigma}}_{1\times Z \; Z\times\Sigma} \frac{\partial \sigma}{\partial \mathbf{W}^1}$$

$$f : \sigma \mapsto \mathbf{z} \quad \text{vector to vector}$$

$$\frac{\partial \mathbf{z}}{\partial \sigma} = \mathbf{W}^{[2]}; \quad \mathbf{z} = \mathbf{W}^{[2]}\sigma + \mathbf{b}^{[2]}$$

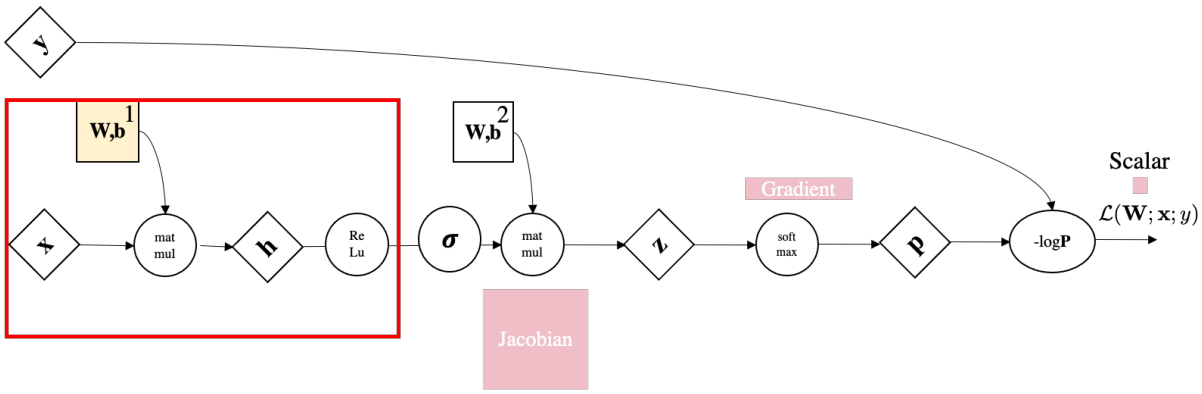


Backprop

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^1} = \underbrace{\frac{\partial \mathcal{L}}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial \sigma}}_{1\times Z \; Z\times\Sigma} \frac{\partial \sigma}{\partial \mathbf{W}^1}$$

$$f : \sigma \mapsto \mathbf{z} \quad \text{vector to vector}$$

$$\frac{\partial \mathbf{z}}{\partial \sigma} = \mathbf{W}^{[2]}; \quad \mathbf{z} = \mathbf{W}^{[2]}\sigma + \mathbf{b}^{[2]}$$

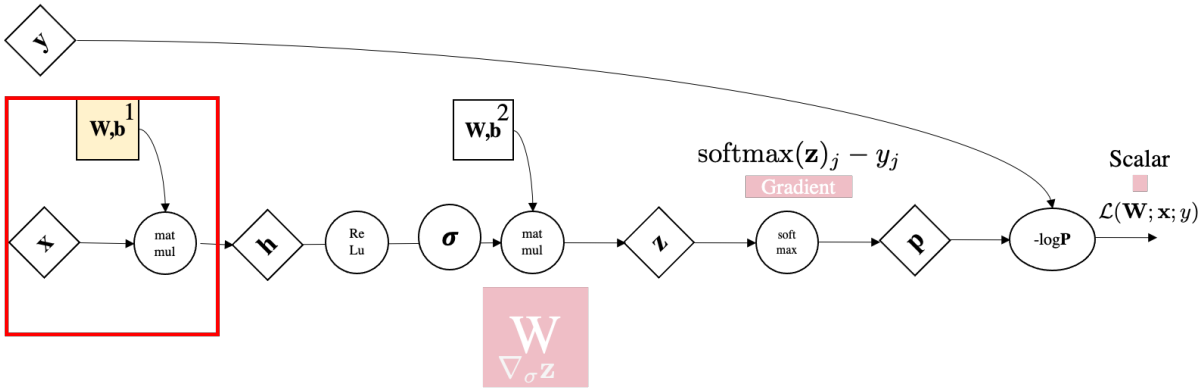


Backprop

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^1} = \underbrace{\frac{\partial \mathcal{L}}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial \sigma}}_{1\times Z \; Z\times\Sigma} \frac{\partial \sigma}{\partial \mathbf{W}^1}$$

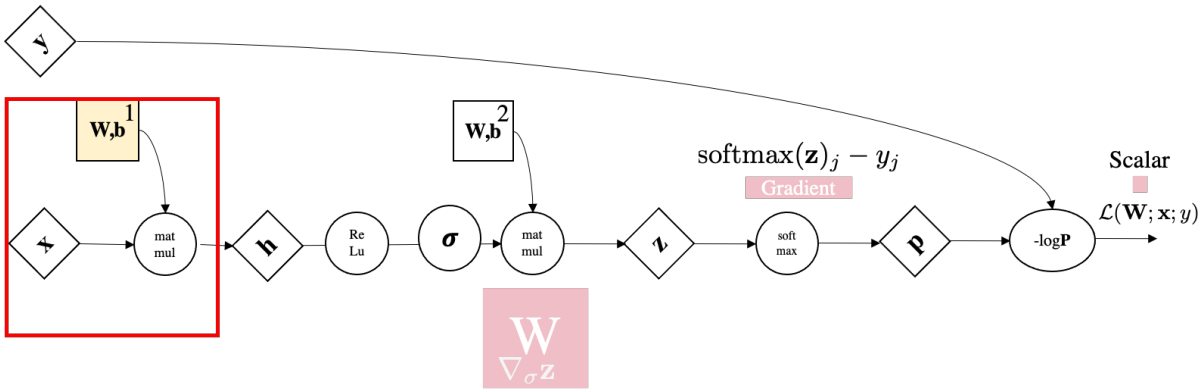
$$f : \sigma \mapsto \mathbf{z} \quad \text{vector to vector}$$

$$\frac{\partial \mathbf{z}}{\partial \sigma} = \mathbf{W}^{[2]}; \quad \mathbf{z} = \mathbf{W}^{[2]}\sigma + \mathbf{b}^{[2]}$$



Backprop

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^1} = \underbrace{\frac{\partial \mathcal{L}}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial \sigma}}_{1\times Z \; Z\times\Sigma} \underbrace{\frac{\partial \sigma}{\partial \mathbf{h}} \frac{\partial \mathbf{h}}{\partial \mathbf{W}^1}}_{\frac{\partial \sigma}{\partial \mathbf{W}^1}}$$

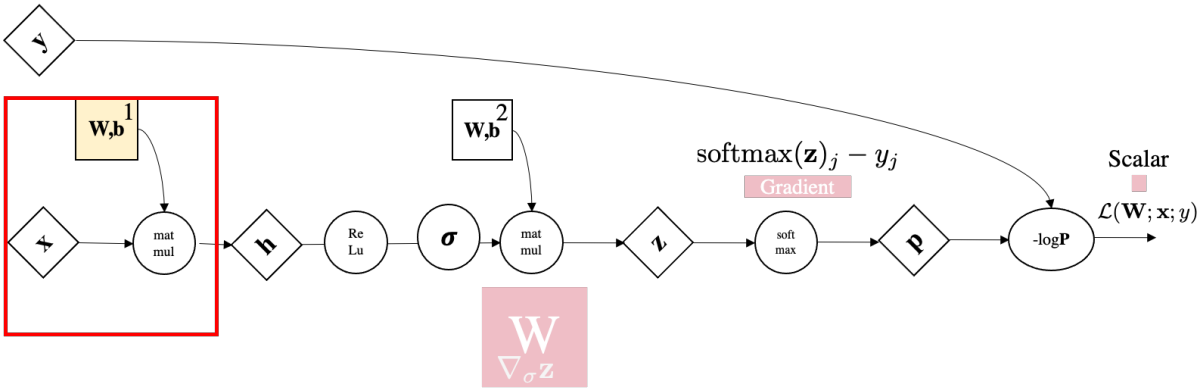


Backprop

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^1} = \underbrace{\frac{\partial \mathcal{L}}{\partial \mathbf{z}}}_{1\times Z}\underbrace{\frac{\partial \mathbf{z}}{\partial \sigma}}_{Z\times\Sigma}\underbrace{\frac{\partial \sigma}{\partial \mathbf{h}}\frac{\partial \mathbf{h}}{\partial \mathbf{W}^1}}_{\frac{\partial \sigma}{\partial \mathbf{W}^1}}$$

$f : \mathbf{h} \mapsto \sigma$ vector to vector - ReLU activation

$$\frac{\partial \sigma}{\partial \mathbf{h}} = \ ?; \quad \mathbf{h}_i = \max(0, \sigma_i) \ \ \forall i \in 0 \dots \Sigma$$

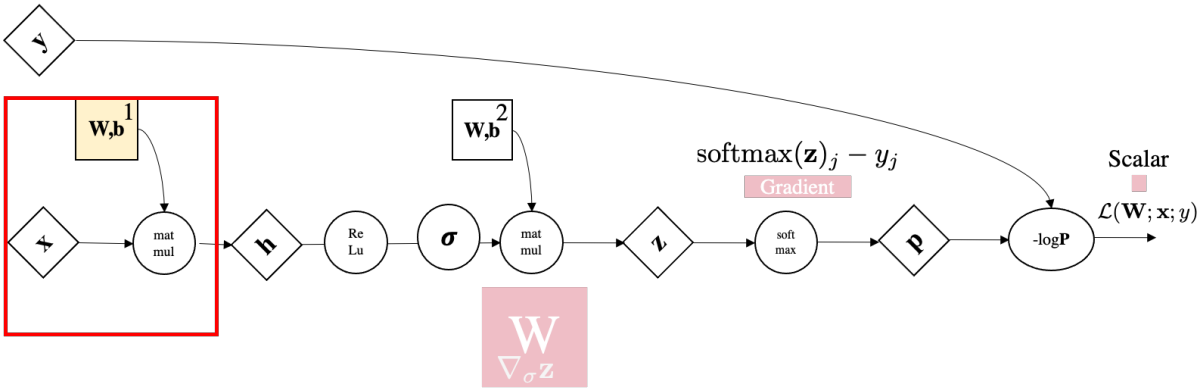


Backprop

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^1} = \underbrace{\frac{\partial \mathcal{L}}{\partial \mathbf{z}}}_{1\times Z}\underbrace{\frac{\partial \mathbf{z}}{\partial \sigma}}_{Z\times\Sigma}\underbrace{\frac{\partial \sigma}{\partial \mathbf{h}}\frac{\partial \mathbf{h}}{\partial \mathbf{W}^1}}_{\frac{\partial \sigma}{\partial \mathbf{W}^1}}$$

$f : \mathbf{h} \mapsto \sigma$ vector to vector **- ReLU activation has a particular Jacobian**;

$$\frac{\partial \sigma}{\partial \mathbf{h}} = \mathrm{diag}(\{0,1\}_{\mathrm{ii}}); \quad \mathbf{h}_i = \max(0, \sigma_i) \ \ \forall i$$

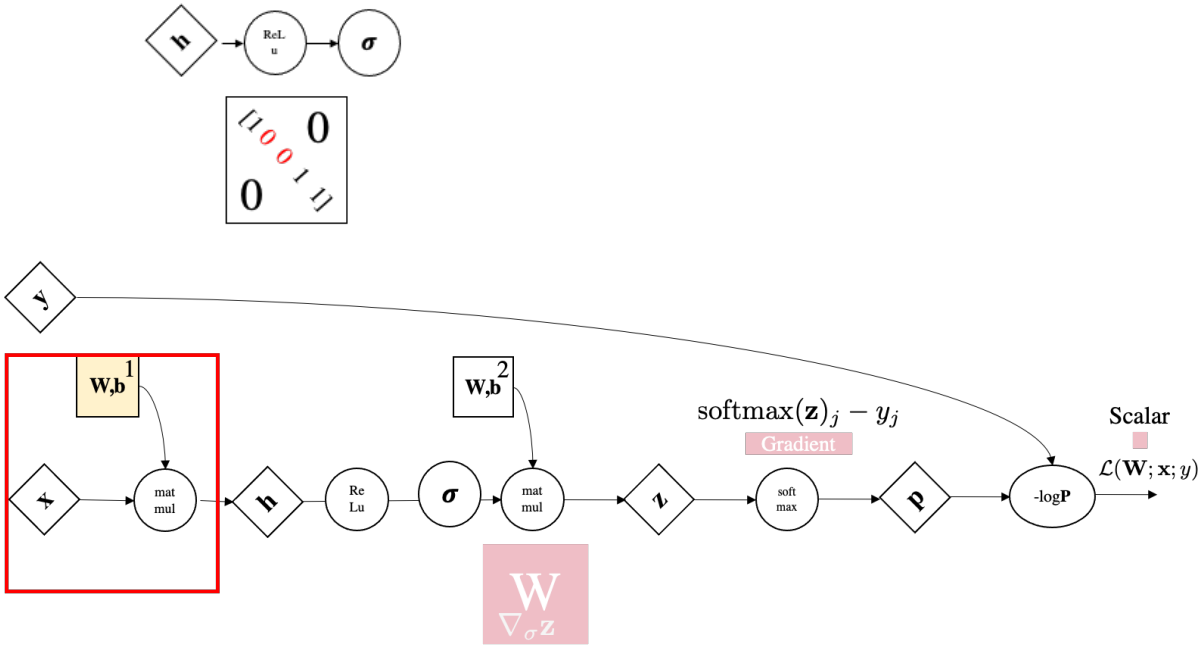


Backprop

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^1} = \underbrace{\frac{\partial \mathcal{L}}{\partial \mathbf{z}}}_{1\times Z}\underbrace{\frac{\partial \mathbf{z}}{\partial \sigma}}_{Z\times \Sigma}\underbrace{\frac{\partial \sigma}{\partial \mathbf{h}}\frac{\partial \mathbf{h}}{\partial \mathbf{W}^1}}_{\frac{\partial \sigma}{\partial \mathbf{W}^1}}$$

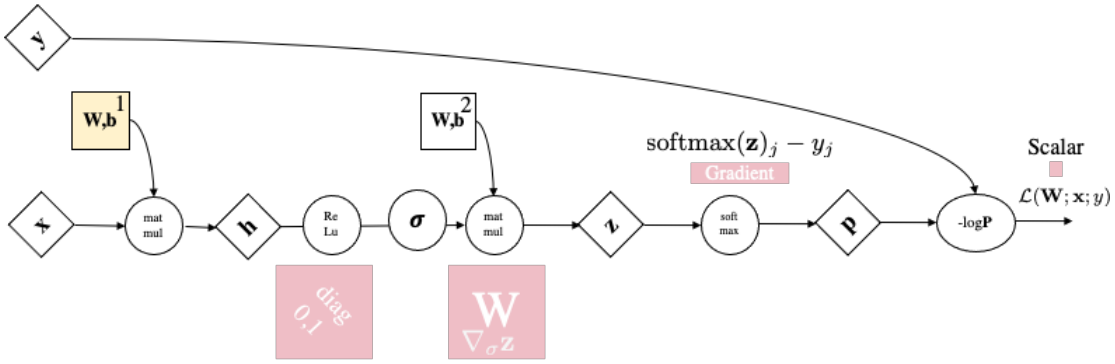
[0.5 -100 -0.1 200 1]

[0.5 0 0 200 1]



Backprop

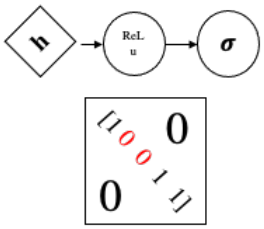
$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^1} = \underbrace{\frac{\partial \mathcal{L}}{\partial \mathbf{z}}}_{1\times Z}\underbrace{\frac{\partial \mathbf{z}}{\partial \sigma}}_{Z\times \Sigma}\underbrace{\frac{\partial \sigma}{\partial \mathbf{h}}}_{\Sigma\times \Sigma}\frac{\partial \mathbf{h}}{\partial \mathbf{W}^1}$$



Jacobian of Activation Functions: implementation exploits sparse structure

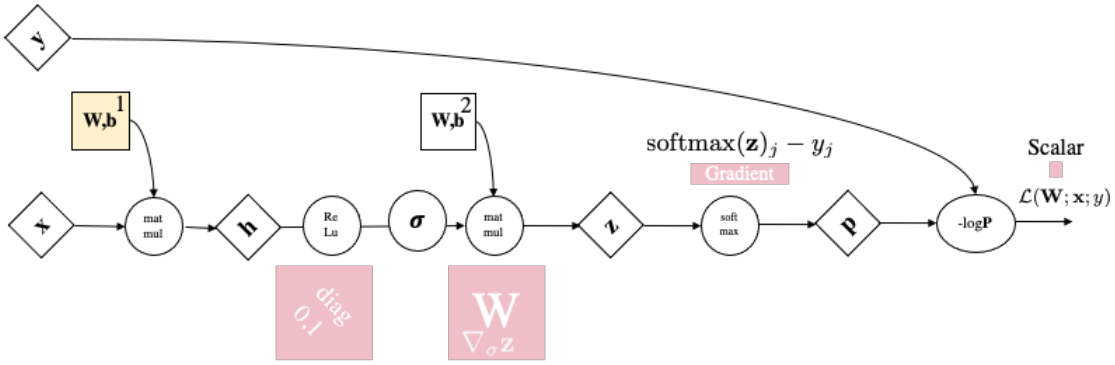
- Jacobians for activation functions are **full of zeros**! Since they are diagonal matrix, they are not implemented as a full matrix in practice but as **the diagonal vector.**
- For a $2^{12} = 4096$ input and $2^{12} = 4096$ output with ReLU, the Jacobian would be of size $2^{24} = 16M$!
- This would scale even worse if you think of a **mini batch of 128 examples:** $2^{12+7} = 4096 \cdot 128$. Jacobian of $2^{(12+7)2} = 274B$!

[0.5 -100 -0.1 200 1] [0.5 0 0 200 1]



Backprop

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^1} = \underbrace{\frac{\partial \mathcal{L}}{\partial \mathbf{z}}}_{1\times Z}\underbrace{\frac{\partial \mathbf{z}}{\partial \sigma}}_{Z\times\Sigma}\underbrace{\frac{\partial \sigma}{\partial \mathbf{h}}}_{\Sigma\times\Sigma}\frac{\partial \mathbf{h}}{\partial \mathbf{W}^1}$$



How to compute $\frac{\partial \mathbf{h}}{\partial \mathbf{W}^{[1]}}$? Matrix to vector function

Matrix notation (I dropped the layer number but $\mathbf{W}$ is $\mathbf{W}^{[1]}$):

$$\mathbf{h} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

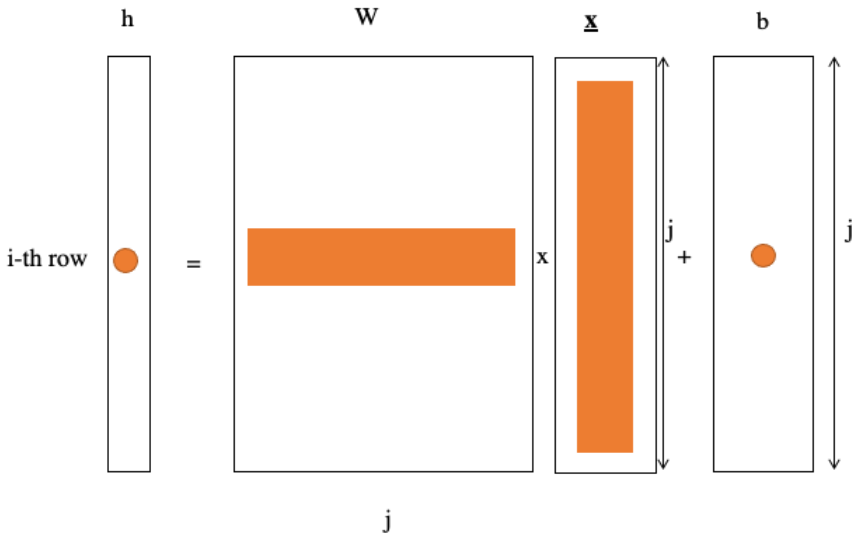
Vector Notation **(slice a single row)** let's consider the **i-th row of** $\mathbf{h}$ so $h = \mathbf{h}_i$:

$$h = \mathbf{w}^T\mathbf{x} + b \quad \text{where } \mathbf{w} \text{ selects i-th row of } \mathbf{W}$$

Slice a single row

$$h = \sum_j w_j x_j + b$$

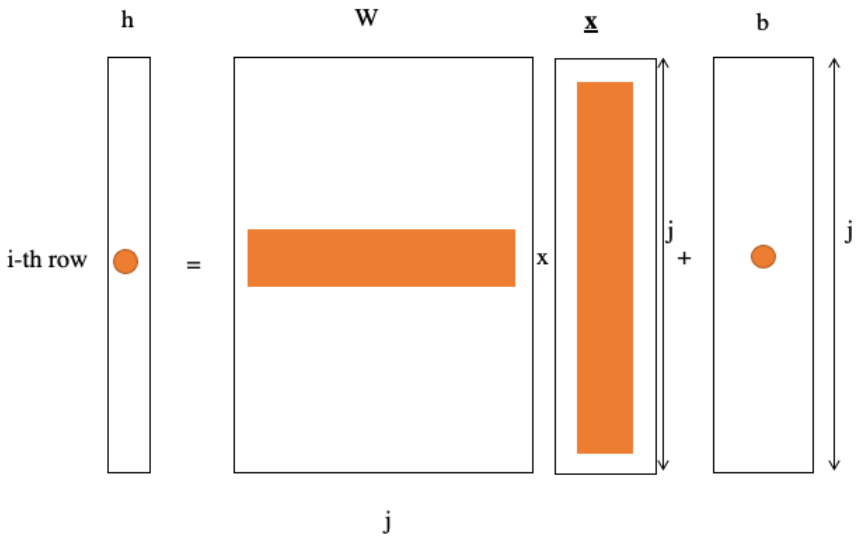
$$\frac{\partial h}{\partial w_j} = x_j$$



Gradient of $\mathbf{h}$ wrt $\mathbf{W}_{ij}$ ($\mathbf{W}_{ij} \mapsto \mathbf{h}$)

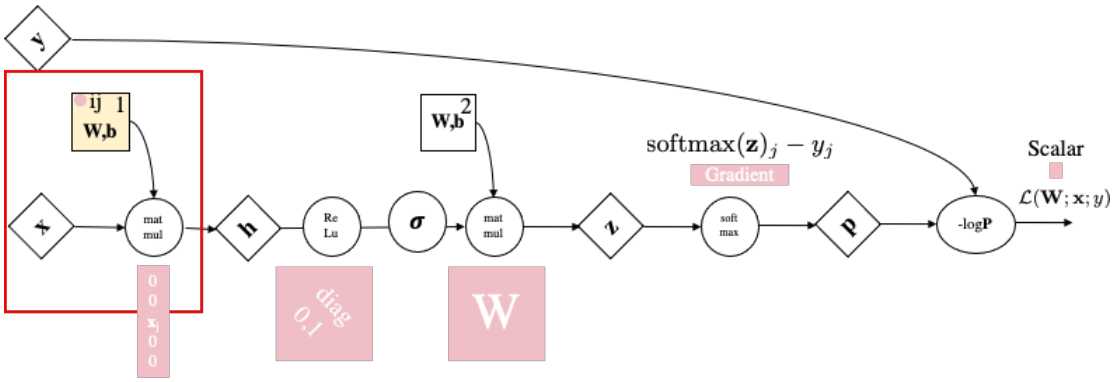
$$\mathbf{h}_i = \sum_j \mathbf{W}_{ij} x_j + \mathbf{b}_i$$

$$\frac{\partial \mathbf{h}}{\partial \mathbf{W}_{ij}} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ x_j \;\; i^{\text{th}}\text{row} \\ \vdots \\ 0 \end{bmatrix}$$

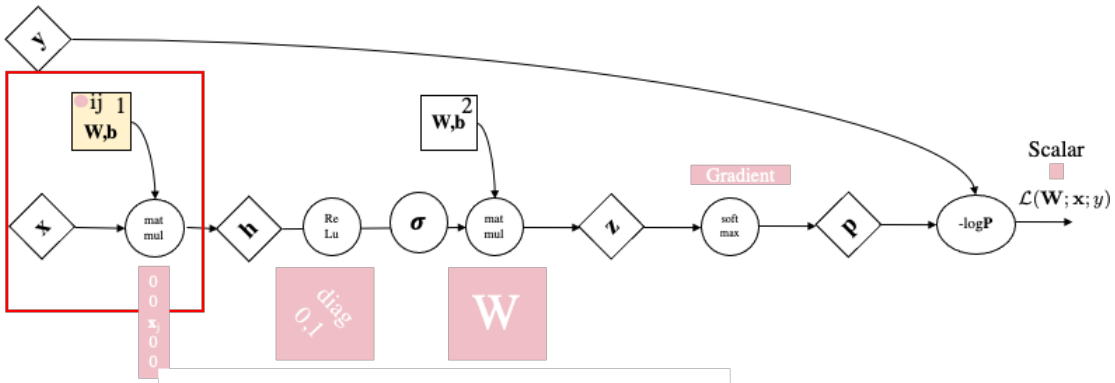


Backprop $\mathcal{L}$ to $\mathbf{W}_{ij}^{[1]}$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{ij}^{[1]}} = \underbrace{\frac{\partial \mathcal{L}}{\partial \mathbf{z}}}_{1\times Z}\underbrace{\frac{\partial \mathbf{z}}{\partial \sigma}}_{Z\times\Sigma}\underbrace{\frac{\partial \sigma}{\partial \mathbf{h}}}_{\Sigma\times\Sigma}\underbrace{\frac{\partial \mathbf{h}}{\partial \mathbf{W}_{ij}^{[1]}}}_{\Sigma\times 1}$$



Backprop $\mathcal{L}$ to $\mathbf{W}_{ij}^{[1]}$



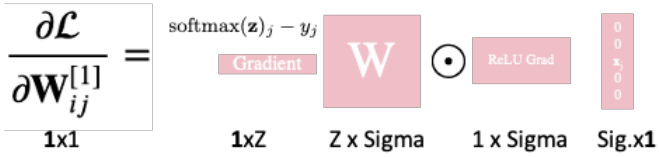
$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{ij}^{[1]}} = \underbrace{\frac{\partial \mathcal{L}}{\partial \mathbf{z}}}_{1\times Z}\underbrace{\frac{\partial \mathbf{z}}{\partial \sigma}}_{Z\times\Sigma}\underbrace{\frac{\partial \sigma}{\partial \mathbf{h}}}_{\Sigma\times\Sigma}\underbrace{\frac{\partial \mathbf{h}}{\partial \mathbf{W}_{ij}^{[1]}}}_{\Sigma\times 1}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{ij}^{[1]}} = \text{softmax}(\mathbf{z})_j - y_j \quad \text{Gradient} \quad \mathbf{W} \quad \text{diag } 0,1 \quad 0\ 0\ x_i\ 0\ 0$$

1x1 1xZ Z x Sigma Sig. x Sig. Sigmax1

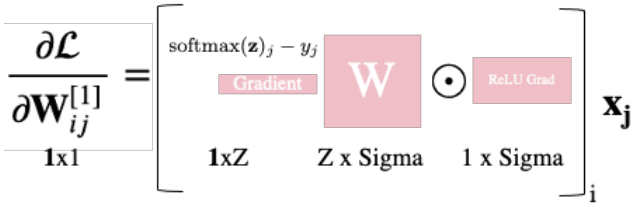
Backprop $\mathcal{L}$ to $\mathbf{W}_{ij}^{[1]}$ simplified

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{ij}^{[1]}} = \underbrace{\frac{\partial \mathcal{L}}{\partial \mathbf{z}}}_{1\times Z}\underbrace{\frac{\partial \mathbf{z}}{\partial \sigma}}_{Z\times\Sigma}\underbrace{\frac{\partial \sigma}{\partial \mathbf{h}}}_{\Sigma\times\Sigma}\underbrace{\frac{\partial \mathbf{h}}{\partial \mathbf{W}_{ij}^{[1]}}}_{\Sigma\times 1}$$



Backprop $\mathcal{L}$ to $\mathbf{W}_{ij}^{[1]}$ simplified

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{ij}^{[1]}} = \underbrace{\frac{\partial \mathcal{L}}{\partial \mathbf{z}}}_{1\times Z}\underbrace{\frac{\partial \mathbf{z}}{\partial \sigma}}_{Z\times\Sigma}\underbrace{\frac{\partial \sigma}{\partial \mathbf{h}}}_{\Sigma\times\Sigma}\underbrace{\frac{\partial \mathbf{h}}{\partial \mathbf{W}_{ij}^{[1]}}}_{\Sigma\times 1}$$



Backprop $\mathcal{L}$ to $\mathbf{W}^{[1]}$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{ij}^{[1]}} = \underbrace{\frac{\partial \mathcal{L}}{\partial \mathbf{z}}}_{1\times Z} \underbrace{\frac{\partial \mathbf{z}}{\partial \sigma}}_{Z\times\Sigma} \underbrace{\frac{\partial \sigma}{\partial \mathbf{h}}}_{\Sigma\times\Sigma} \underbrace{\frac{\partial \mathbf{h}}{\partial \mathbf{W}_{ij}^{[1]}}}_{\Sigma\times 1}$$

Outer Product

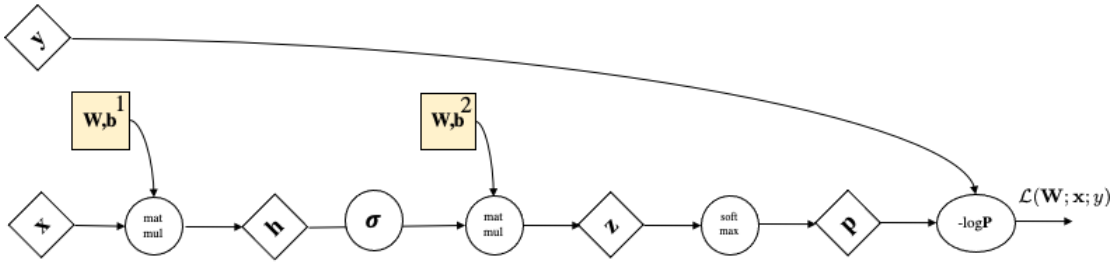
$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{[1]}} = \left[\underbrace{\text{softmax}(\mathbf{z})_j - y_j}_{\text{Gradient}} \; \mathbf{W} \odot \underbrace{\text{ReLU Grad}} \right] \mathbf{x}^{\mathbf{T}}$$

$$\square = \square \; \square$$

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

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

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

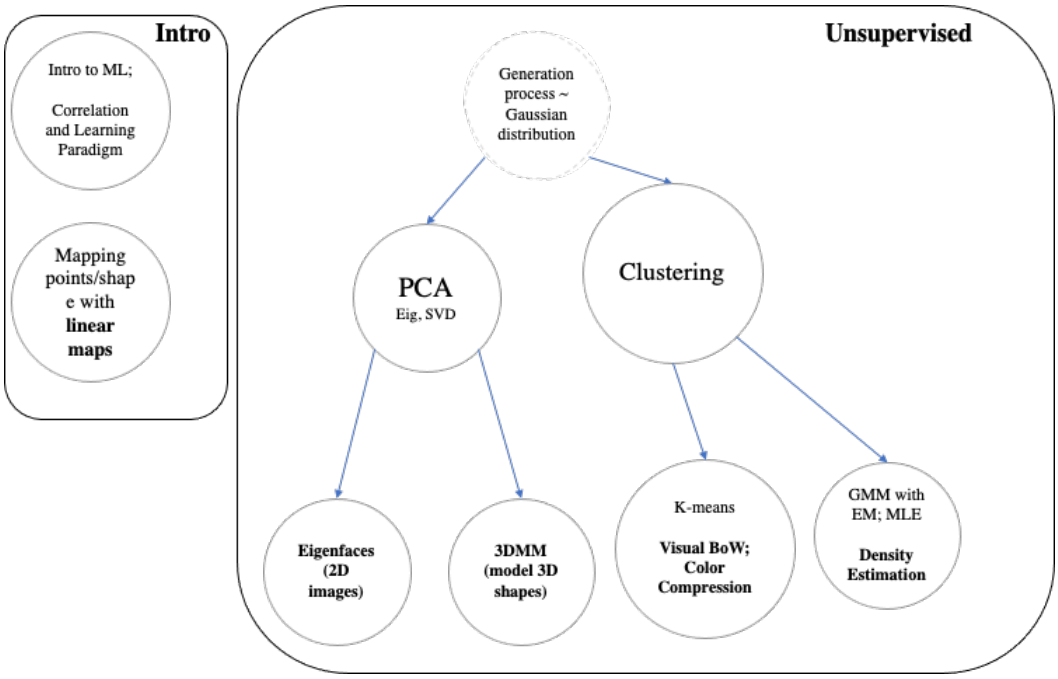


A look back to the Entire Course

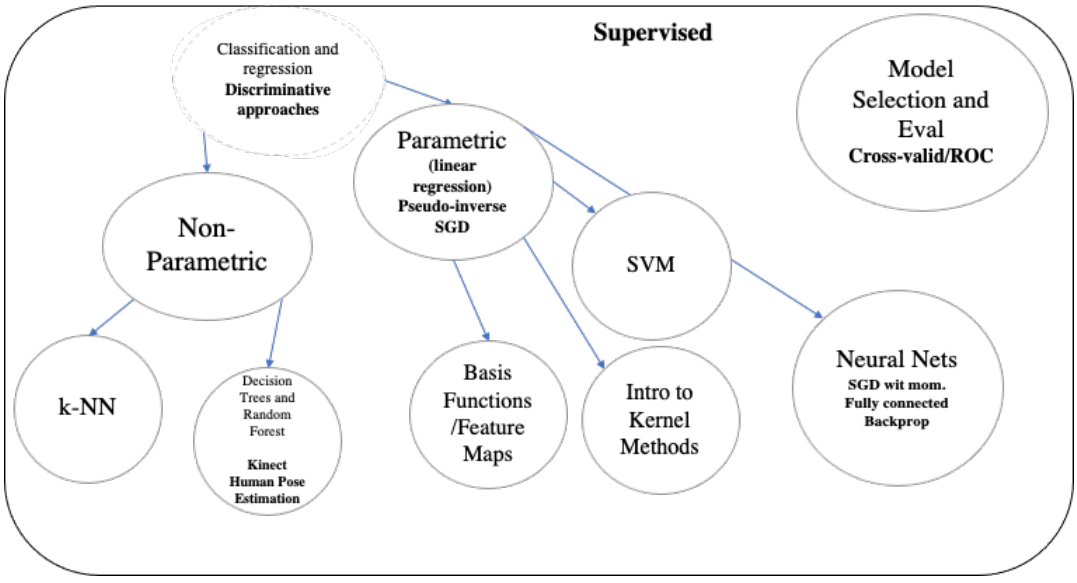
AI & ML are interesting with striking applications but price to pay...

- *Solid math background* (probability, statistics, linear algebra, calculus)
- Computer scientist formulates/interprets math and knows what is best to turn it into computational system
- Have critical minds on how to evaluate ML systems

A look back to the Entire Course



A look back to the Entire Course



🏁END of the LINE 🏁

Soon it will be your turn on 13 June 2023

Do not worry there are also

- Exam session on **6 July 2023**
- Exam session on **14 September 2023**

Million Dollar 🤑 link

How to prepare to the exam

1. Simple **exercises** like the ones we saw for Decision Trees, KNN etc. but also on other methods (SVM, linear regression, SoftMax etc.)
2. **Definitions + Proof Sketch** like the ones that we show in class.
3. Answering critical open questions but you have to show **some kind of math or graph/plot sketch behind your rationale**

- Still deciding about adding `multiple choice questions`
- Questions will be spread over the entire program *(math, norms, clustering, supervised non-parametric, parametric models, neural nets etc)*.

The questions will be of increasing difficulties and calibrated to the time you have.

That's all folks! 👋🏻