Pytorch Tutorials

Gradient Calculation.

{ PyTorch uses the chain rule in order to calculate the gradients. }

· Jacobian: $J = \begin{bmatrix} \nabla^T f_1 \\ \vdots \\ \nabla^T f_m \end{bmatrix} = \overset{m \times n}{\begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}}$

forward (calculate the «ys»)

$\Rightarrow$ $x, 2 \rightarrow \oplus \rightarrow y$

backward $\boxed{\frac{\partial y}{\partial x}}$

$\boxed{x \longrightarrow y \longrightarrow z \longrightarrow L}$

Example

$L = \frac{z_1 + z_2}{3}$, $\vec{z} = 2\vec{y}^2$, $\vec{y} = \vec{x} + 2$ $\Rightarrow$ suppose we have 2 features.

$\Rightarrow \underset{\mathbb{R}^{1\times 2}}{\frac{\partial L}{\partial x}} = \underset{\mathbb{R}^{1\times 2}}{\frac{\partial L}{\partial z}} \cdot \underset{\mathbb{R}^{2\times 2}}{\frac{\partial z}{\partial y}} \cdot \underset{\mathbb{R}^{2\times 2}}{\frac{\partial y}{\partial x}}$

Backward pass $\Rightarrow$

① $\begin{bmatrix} \frac{\partial L}{\partial z_1} & \frac{\partial L}{\partial z_2} \end{bmatrix} = \begin{bmatrix} \frac{1}{3} & \frac{1}{3} \end{bmatrix}$

② $\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial y}$

$= \underset{1 \times 2}{\begin{bmatrix} \frac{1}{3} & \frac{1}{3} \end{bmatrix}} \underset{2 \times 2}{\begin{bmatrix} \frac{\partial z_1}{\partial y_1} & \frac{\partial z_2}{\partial y_2} \\ \frac{\partial z_2}{\partial y_2} & \frac{\partial z_2}{\partial y_2} \end{bmatrix}} = \begin{bmatrix} \frac{4y_1}{3} \\ \frac{4y_2}{3} \end{bmatrix}$

③ $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial x}$ = m̂ chose puisque $\frac{\partial y}{\partial x} \Rightarrow Id$

If we have $x_1 = 1$ and $x_2 = 2$:

$\frac{\partial L}{\partial x} = \begin{bmatrix} \frac{4(3)}{2} \\ \frac{4(4)}{2} \end{bmatrix} = \begin{bmatrix} 6 \\ 8 \end{bmatrix}$ $\Rightarrow$ le gradient final.

$\boxed{\frac{\partial L}{\partial y} = \underset{\boxed{\text{Vector}}}{\frac{\partial L}{\partial z}} \cdot \frac{\partial z}{\partial y}}$ $\Rightarrow$ $\boxed{\text{Jacobian}}$

In Pytorch

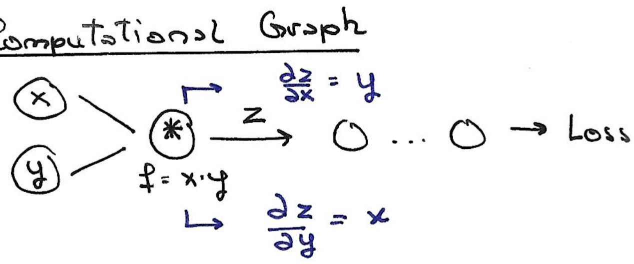
- → Prediction: PyTorch Model
- → Gradients Computation: Autograd
- → Loss Computation: Pytorch Loss
- → Parameter Updates: Pytorch Optimizer

Backpropagation.

Chain Rule: $\frac{\partial z}{\partial x} = \underset{①}{\frac{\partial z}{\partial y}} \cdot \underset{②}{\frac{\partial y}{\partial x}}$

$x \to \underset{②}{a(x)} \xrightarrow{y} \underset{①}{b(y)} \to z$

Computational Graph



LOCAL GRADIENTS.

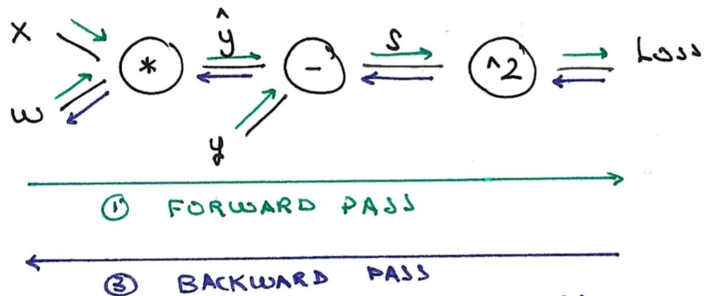
{ we wish to calculate $\frac{\partial Loss}{\partial x} = ?$ }

$$\frac{\partial Loss}{\partial x} = \frac{\partial Loss}{\partial z} \cdot \frac{\partial z}{\partial x}$$

BACKPROPAGATION ALGORITHM

1) Forward pass: Compute the loss
2) Compute Local Gradients
3) Backward pass: Compute $\frac{\partial Loss}{\partial weights}$ using the chain rule.

Example: $\boxed{\hat{y} = wx}$ $\boxed{loss = (\hat{y} - y)^2 = (wx - y)^2}$



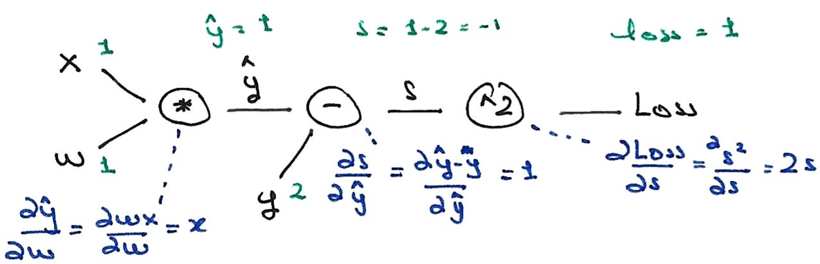
② LOCAL GRADIENTS: $\frac{\partial \hat{y}}{\partial w}$, $\frac{\partial s}{\partial \hat{y}}$, $\frac{\partial Loss}{\partial s}$

$\boxed{\frac{\partial Loss}{\partial w}} \leftarrow \frac{\partial Loss}{\partial \hat{y}} \leftarrow \frac{\partial Loss}{\partial s}$

with a training sample: $x = 1$, $y = 2$, $w = 1$

$\Rightarrow$ Therefore for this specific sample:

$$\left\{ \frac{\partial Loss}{\partial w} = -2 \right\}$$



$$\frac{\partial Loss}{\partial s} = 2s \Rightarrow \frac{\partial Loss}{\partial \hat{y}} = \frac{\partial Loss}{\partial s} \cdot \frac{\partial s}{\partial \hat{y}} = 2s \cdot 1 = -2$$

$$\Rightarrow \frac{\partial Loss}{\partial w} = \frac{\partial Loss}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w} = -2 \cdot x = \boxed{-2}$$

CNN

→ Convolutional Layer
→ Max-Pooling (reduce the size of the image)
↳ computational cost, avoid over-fitting as well.

Datasets and Dataloaders

- Epoch: 1 forward and backward pass of ALL training samples
- Batch Size: number of training samples in one forward ⊕ backward pass
- Number of iterations: passes ⇒ math.ceil (total-samples / batch size)

- Dataset: stores the samples and their corresponding labels.
- Dataloader: wraps an iterable around the Dataset to enable easy access to the samples.

Transforms in Pytorch.

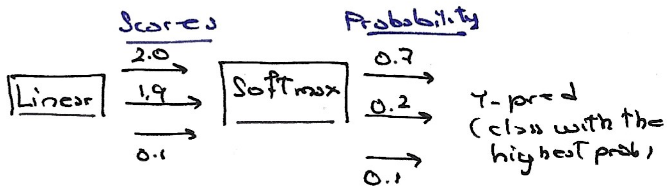
Transforms
→ On Images: Crop. Grayscale. Pad, Random Rotation, etc.
→ On Tensors: Linear Transformation, Normalize. Random Erasing.
→ Conversion: To PILImage: from tensor or ndarray.
To Tensor: from numpy.ndarray or PILImage

→ Generic: Lambda
→ Custom classes / compose multiple transforms.

Ex: composed = transforms. Compose ([Rescale(256), Random Crop(224)])
↳ Torchvision. transforms. Compose ([List of transformations])

Softmax and Cross-Entropy.

- Softmax function: $S(y_i) = \frac{e^{y_i}}{\sum e^{y_i}}$



- Cross-Entropy: loss function for multi-class classification

$D(Y, \hat{Y}) = -\frac{1}{N} \cdot \sum y_i \log(\hat{y}_i)$

↳ Note: $\boxed{\begin{array}{l} Y\text{: must be one-hot encoded} \\ \hat{Y}\text{: must be softmax probabilities} \end{array}}$

$\boxed{loss(\hat{Y}, Y)}$

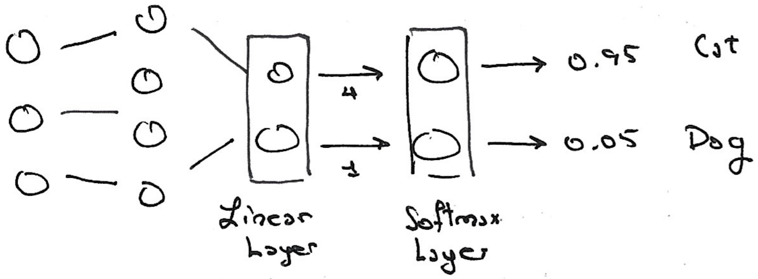
In Pytorch...
nn.crossEntropyLoss applies nn.logsoftmax + nn.NLLLoss
(neg. likelyhood loss)

⇒ There fore, NO SOFTMAX layer!

⇒ Y: not One-hot! → class labels
⇒ $\hat{Y}$: no softmax! → raw scores

Neural Net with Softmax

Which Animal? Multi-Class Classification Problem



{ In Pytorch: no softmax at the end since we're using: nn.CrossEntropyLoss() }

→ If binary classification. we use sigmoid function with nn.BCELoss()

↳ Here. we must calculate the sigmoid in the forward function.

[BINARY → sigmoid (1 output) → nn.BCELoss() → writes sigmoid in forward function

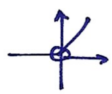
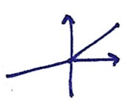
MULTI-CLASS → softmax → nn.CrossEntropyLoss() → ∅ write softmax in forward function (nn method calculates the softmax for us)]

Activation Functions → decide if the neuron should be activated or not!

- without AF, our network is basically just a stacked linear reg. model.

After each layer, we typically use an AF?

Most popular AF:

- Step function → $f(x) = \mathbb{1}\{x \geq \theta\}$
- Sigmoid → $f(x) = \frac{1}{1+e^{-x}}$ 
- TanH ↳ $f(x) = \frac{2}{1+e^{-2x}} - 1$  (good choice for hidden layers)
- ReLU → $f(x) = \max(0, x)$  (good for hidden layers) ← use this if no choice.
- Leaky ReLU → $f(x) = \begin{cases} x & \text{if } x \geq 0 \\ a \cdot x \end{cases}$  (solve vanishing gradient problem)
- Softmax

↓

$S(y_i) = \frac{e^{y_i}}{\sum e^{y_i}}$

(multi-class problem)

⇓

last layer.

Feed-Forward Neural Network

Dataset → Dataloader Transformation → NN + AF → Loss Optimizer → Training Loop (Batch Training) → Model Evaluation → GPU Support