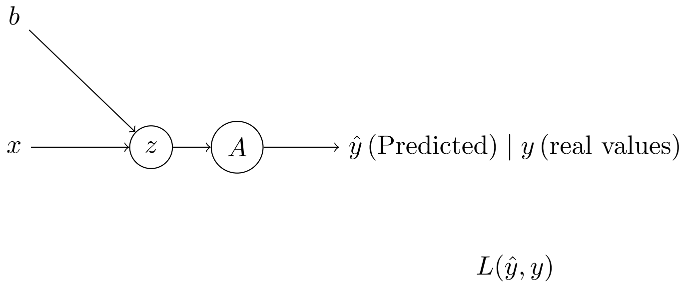
Ramtin Behesht Aeen ramtinba145822@gmail.com

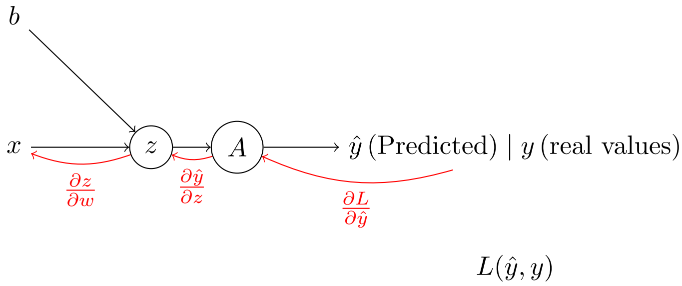
Summary of

1 Neural Network Theory

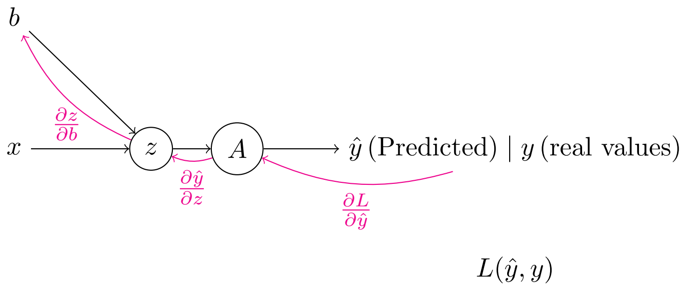
Neural Network Structure:



backward pass for w:



backward pass for b:



backward pass for w mathematically:

$$L = \underbrace{L(\underbrace{\hat{y}\,(\underbrace{z(Xw+b)}_{\frac{\partial z}{\partial w}})}_{\frac{\partial \hat{y}}{\partial z}})}_{\frac{\partial L}{\partial \hat{y}}} \tag{1}$$

$$\frac{\partial L}{\partial w} = \underbrace{\frac{\partial L}{\partial z}}_{\frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z}} \frac{\partial z}{\partial w} \tag{2}$$

as a Summary:

$$L = \underbrace{L\,(\hat{y})}_{\frac{\partial L}{\partial \hat{y}}} = \underbrace{L\,(\hat{y}\,(z))}_{\frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z}} = \underbrace{L\,(\hat{y}\,(z(XW+b)))}_{\frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial W}}$$

Or more explicitly for the chain rule:

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial z} \times \frac{\partial z}{\partial W}$$

- $z = XW + b$
- $\hat{y} = f(z)$ (activation function, e.g. softmax)
- $L = loss(\hat{y}, y)$

The derivatives $\frac{\partial L}{\partial \hat{y}}$, $\frac{\partial \hat{y}}{\partial z}$, and $\frac{\partial z}{\partial W}$ represent the gradient chain from output all the way down to weights.

2 Coding the Neural Network

Neural Network Structure:

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial w}$$

where:

- **Loss derivative**: $\frac{\partial L}{\partial \hat{y}}$
- **Activation derivative**: $\frac{\partial \hat{y}}{\partial z}$
- **Linear model derivative**: $\frac{\partial z}{\partial w}$

1. Loss derivative $\left(\frac{\partial L}{\partial \hat{y}}\right)$: This is computed in:

```python
def backward(self) -> np.ndarray:  # CrossEntropy
    grad = -self.target / self.prediction / self.target.shape[0]
    return grad
```

- `self.prediction` $= \hat{y}$ (output of softmax)
- This gives the gradient from the loss with respect to the softmax output $\hat{y}$.
- This matches step 1 of the professor's chain rule.

Activation derivative $\left(\frac{\partial \hat{y}}{\partial z}\right)$: If you have a `Softmax` layer:

```python
def backward(self, up_grad: np.ndarray) -> np.ndarray:  # Softmax
    ...
    down_grad[i] = np.dot(jacobian, up_grad[i])
```

- `up_grad` is $\frac{\partial L}{\partial \hat{y}}$ from the loss.
- The softmax Jacobian gives $\frac{\partial \hat{y}}{\partial z}$.
- Output `down_grad` is $\frac{\partial L}{\partial z}$.
- This matches step 2 of the chain rule.

Linear model derivative $\left(\frac{\partial z}{\partial w}\right)$: In your `Linear` layer:

```python
def backward(self, up_grad: np.ndarray) -> np.ndarray:  # Linear
    self.dw = np.dot(self.inp.T, up_grad)  # ∂L/∂w
    self.db = np.sum(up_grad, axis=0, keepdims=True)  # ∂L/∂b
    down_grad = np.dot(up_grad, self.w.T)  # ∂L/∂input
    return down_grad
```

- `up_grad` is $\frac{\partial L}{\partial z}$ from the activation.
- Multiplying with `inp.T` applies $\frac{\partial z}{\partial w}$ to get $\frac{\partial L}{\partial w}$.
- This is step 3 of the professor's chain rule.

Full Chain in Code:

1. Loss backward → `CrossEntropy.backward()`

$$\frac{\partial L}{\partial \hat{y}}$$

2. Activation backward → `Softmax.backward()`

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

3. Linear backward → `Linear.backward()`

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial w}$$

Linear : Abstract Python Code for Layer:

```python
class Layer:
  def __init__(self):
      self.inp = None
      self.out = None

  def __call__(self, inp: np.ndarray) -> np.ndarray:
      return self.forward(inp)

  def forward(self, inp: np.ndarray) -> np.ndarray:
      raise NotImplementedError

  def backward(self, up_grad: np.ndarray) -> np.ndarray:
      raise NotImplementedError

  def step(self, lr: float) -> None:
      pass
```

- Input features:

$$a \tag{3}$$

- Features weights:

$$a \tag{4}$$

- Bias term:

$$a \tag{5}$$

- Activation function :

$$a \tag{6}$$

- Output of the neuron:

$$y \tag{7}$$

```python
def greet(name):
    print(f"Hello, {name}!")

greet("World")
```

Squared Error(SE): Most Common error function in linear regression is:

$$SE : (y^{(i)} - h(x^{(i)}, w))^2 \tag{8}$$

Sum of Squared Errors (SSE): Cost function should measure all predictions. Thus a choice could be Sum of Squared Error(SE)

$$SSE : \sum_{i=1}^{N} (y^{(i)} - h(x^{(i)}, w)) \tag{9}$$

Solve it analytically for one dimension: Predicted:

$$\widehat{y} = w_0 + w_1 x \tag{10}$$

SSE or Cost Function:

$$J(w_0, w_1) := \sum_{i=1}^{N}(y^{(i)} - \widehat{y}^{(i)})^2 = \sum_{i=1}^{N}(y^{(i)} - w_0 + w_1 x^{(i)})^2 \tag{11}$$

Assumptions:

$$\frac{\partial J}{\partial w_0} = 0\,, \frac{\partial J}{\partial w_1} = 0 \tag{12}$$

$\frac{\partial J}{\partial w_0} = 0$: thus:

$$\frac{\partial}{\partial w_0}\left(\sum_{i=1}^{N}(y^{(i)} - (w_0 + w_1 x^{(i)}))^2\right) = 0\ ^{1} \tag{13}$$

$$-2\sum_{i=1}^{N}(y^{(i)} - w_0 - w_1 x^{(i)}) = 0 \tag{14}$$

For this equation to equal zero, the following condition must be met:

- $\sum_{i=1}^{N} y^{(i)} = 0 := Y$
- $\sum_{i=1}^{N} -w_0 = 0 := nw_0$
- $\sum_{i=1}^{N} -w_1 x^{(i)} = 0 := X$

Thus:

$$0 = Y - nw_0 - w_1 x^{(i)} \longrightarrow w_0 = \frac{(Y - w_1 x^{(i)})}{n} \tag{15}$$

[1] $fog(x)' = f(g(x))' g(x)'$