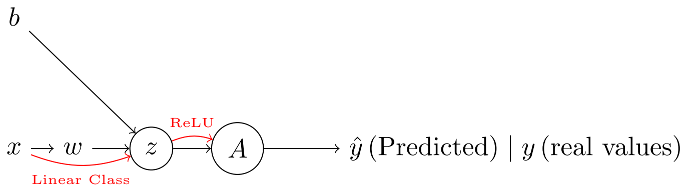
no column mode

Ramtin Behesht Aeen ramtinba145822@gmail.com

Summary of

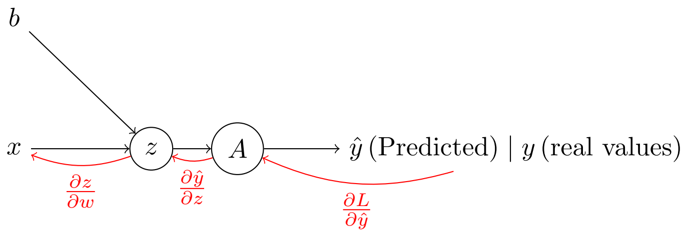
1 Neural Network Theory

Neural Network Structure:



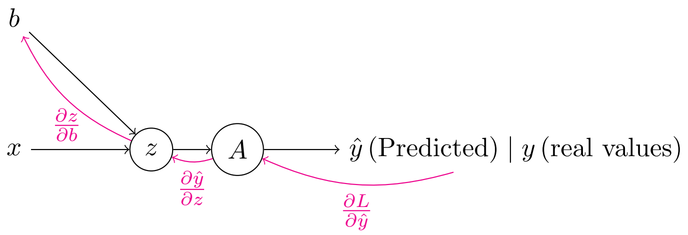
$$L(\hat{y}, y)$$

backward pass for w:



$$L(\hat{y}, y)$$

backward pass for b:



$$L(\hat{y}, y)$$

backward pass for w mathematically:

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

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

as a Summary:

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

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:

```
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 $\hat{y}$.

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

```
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:
$$y \tag{7}$$

explaining Forward in Linear and ReLU:
in Linear and forward class we have:

```
class Linear(Layer):
    ...
    def forward(self, inp: np.ndarray) -> np.ndarray:
        """Perform the linear transformation: output = inp * W + b"""
        self.inp = inp
        self.out = np.dot(inp, self.w) + self.b
        return self.out
  ...
```

and in ReLU we have:

```
class ReLU(Layer):
    def forward(self, inp: np.ndarray) -> np.ndarray:
        """ReLU Activation function: f(x) = max(0, x)"""
        self.inp = inp
        self.out = np.maximum(0, inp)
        return self.out

    def backward(self, up_grad: np.ndarray) -> np.ndarray:
        """Backward pass for ReLU: derivative is 1 where input > 0, else 0."""
        down_grad = up_grad * (self.inp > 0) # Efficient boolean indexing
        return down_grad
```

then we make a list of layers like this:

```
layers = [
    Linear(input_size, hidden_size),
    ReLU(),
    Linear(hidden_size, output_size)
]
#passing layers to MLP Class:
model = MLP(layers, CrossEntropy(), learning_rate=0.001)
```

and this list of layers will be called in the MLP class as following:

```
class MLP:
    def __init__(self, layers: list[Layer], loss_func: Loss, learning_rate: float) -> None:
        self.layers        = layers
        self.loss_func     = loss_func
        self.learning_rate = learning_rate

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

    def forward(self, input: np.ndarray) -> np.ndarray:
        """ pass input through each layer sequentially """
        for layer in self.layers:
            input = layer.forward(input)
        return input
```

so the list of layers will be called one by one and the output of each layer will be feed to the next layer as input. the next simple example will show how it works:

simple example of forward pass:

```python
import numpy as np
# Input (batch of 2 samples, each with 3 features)
X = np.array([[1, 2, 3],
              [4, 5, 6]])

# Linear layer
class Linear:
    def __init__(self, in_features, out_features):
        self.W = np.ones((in_features, out_features)) # just ones for simplicity
        self.b = np.zeros((1, out_features))

    def forward(self, x):
        z = x @ self.W + self.b
        print("Linear output z =", z)
        return z

# ReLU layer
class ReLU:
    def forward(self, x):
        a = np.maximum(0, x)
        print("ReLU output a =", a)
        return a

# --- Build network ---
layers = [Linear(3, 2), ReLU()]

# --- Forward pass ---
out = X
for layer in layers:
    print(out)
    out = layer.forward(out)
    print("After layer forward, out =", out)

print("Final output =", out)
```

First X is set to $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$, W is set to ones, and b is set to zeros.

1. In `out = layer.forward(out)` the first layer is Linear, and `out` is equal to X. The matrix X will be fed into the Linear `forward` function.

2. In the Linear `forward` function: the output z will be calculated as:

$$z[0] = [1 + 2 + 3,\ 1 + 2 + 3] = [6, 6], \quad z[1] = [4 + 5 + 6,\ 4 + 5 + 6] = [15, 15]$$

3. In this step, `out` will be equal to the output of `layer.forward(out)`, which is z.

4. Now in the next step of the loop, z will be fed to the ReLU `forward` function, which is

$$a = \max(0, x)$$

   and it will be equal to

$$\begin{bmatrix} 6 & 6 \\ 15 & 15 \end{bmatrix}$$

5. Then the result will be printed as:

```
Final output = [[6.  6.]  [15.  15.]]
```