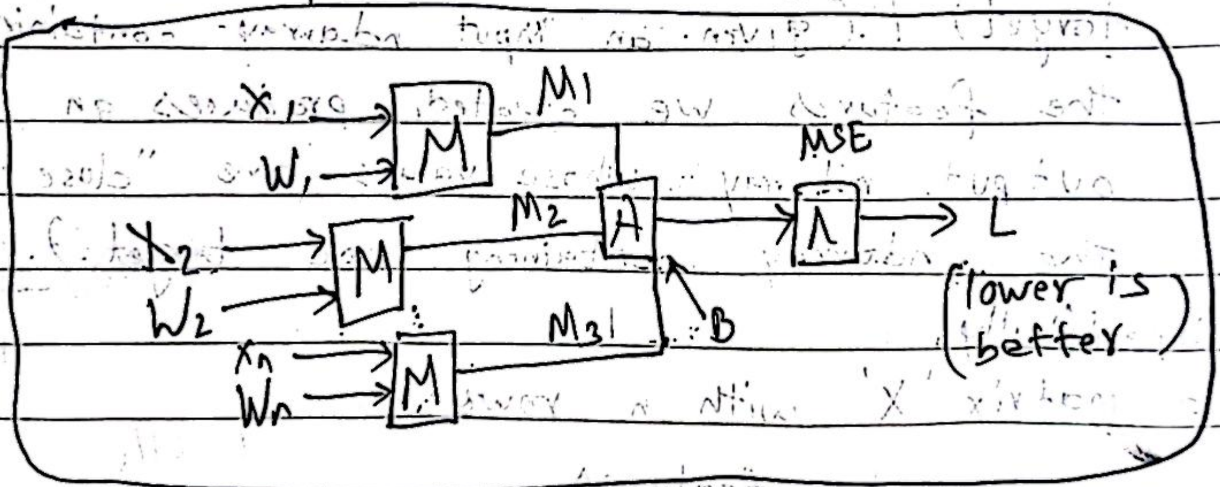
takes X_{batch} & produce vectors of values p_i = "predictions" which are "close to the target values" y_i

Linear Regression

$$y_i = \beta_0 + \beta_1 \times x_1 + \ldots + \beta_n \times x_n + \epsilon$$

Numeric value of each target is a linear combination of the k features of X, plus the β_0 term to adjust the "baseline"



Suppose $X = [x_1 \; x_2 \; \ldots \; x_k]$

$$W = \begin{bmatrix} W_1 \\ W_2 \\ \vdots \\ W_k \end{bmatrix}$$

$p_i = X_i \times W \Rightarrow W_1 x_1 + W_2 x_2 + \ldots + W_k x_k$

"generating the prediction" for LR using: the Dot Product for vectors

matrix multiplication for matrix
For a batch of size 3.

$$X_{batch} = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1k} \\ x_{21} & x_{22} & \dots & x_{2k} \\ x_{31} & x_{32} & \dots & x_{3k} \end{bmatrix}$$

$$P_{batch} = X_{batch} \times W = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1k} \\ x_{21} & x_{22} & \dots & x_{2k} \\ x_{31} & x_{32} & \dots & x_{3k} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_k \end{bmatrix}$$

$$= \begin{bmatrix} x_{11} w_1 + x_{12} w_2 + \dots + x_{1k} w_k \\ x_{21} w_1 + x_{22} w_2 + \dots + x_{2k} w_k \\ x_{31} w_1 + x_{32} w_2 + \dots + x_{3k} w_k \end{bmatrix} = \begin{bmatrix} p_1 \\ p_2 \\ p_3 \end{bmatrix}$$

"Training" this model.
takes data, combine them with parameters in some way, & produce predictions.
→ X as data, W as parameter, p_{batch} prediction using "Matrix multiplication"

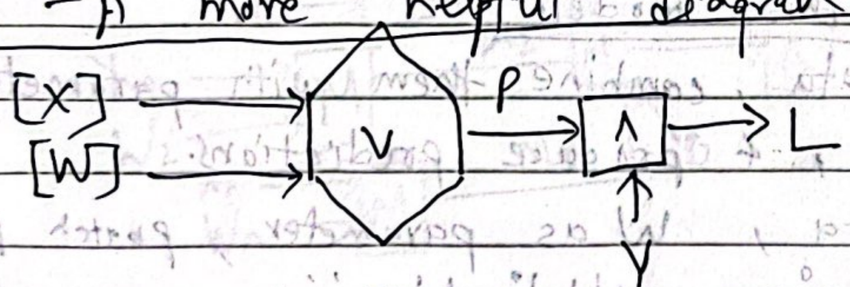
whether or not these observations are good.
We use targets y_{batch} associated with the batch of observations X_{batch} fed into the function, and we compute a single number, function of y_{batch} & p_{batch}. "penalty". Eg: "Mean Squared Error"

$$MSE(P_{batch}, Y_{batch}) = MSE\left(\begin{bmatrix} p_1 \\ p_2 \\ p_3 \end{bmatrix}, \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}\right)$$

$$= \frac{(y_1 - p_1)^2 + (y_2 - p_2)^2 + (y_3 - p_3)^2}{3}$$

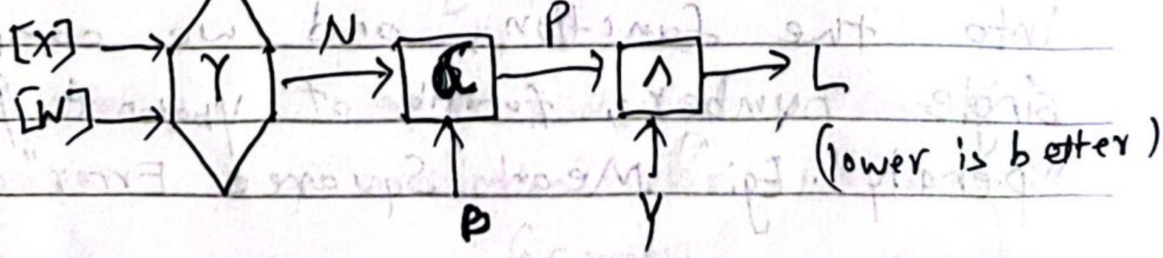
We use all techniques from chapter 1 to compute the "gradient" of this number with respect to W.
Then we can use these derivatives to update each element of W in the direction that would cause L to "decrease".
Repeat → "train"

LR - A more helpful diagram



$$L = \Lambda(\nu(X, W), Y)$$

Adding the intercept in



$3\times2 \quad 3\times3$

$P_{batch\_with\_bias} = X_i \text{ dot } W + b$

$$= \begin{bmatrix} x_{11}W_1 + x_{12}W_2 + \dots + x_{1k}W_k + b \\ x_{21}W_1 + x_{22}W_2 + \dots + x_{2k}W_k + b \\ x_{31}W_1 + x_{32}W_3 + \dots + x_{3k}W_k + b \end{bmatrix} = \begin{bmatrix} P_1 \\ P_2 \\ P_3 \end{bmatrix}$$

"the same number" ~~is~~ should get added as bias.
y-intercept

LR - the code

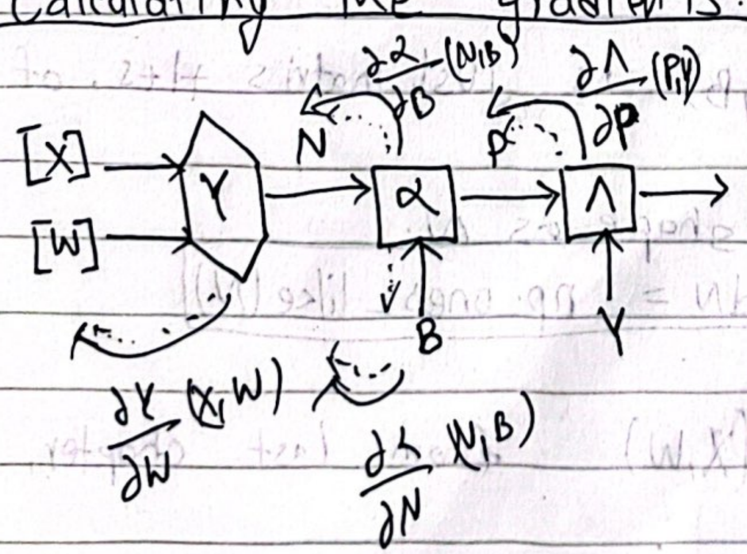
makes prediction & computes losses given batches of observations X_{batch} & their corresponding targets Y_{batch}.

~~#~~Training the Model

compute $\frac{\partial L}{\partial W_i}$ for every w_i in W,

as well as $\frac{\partial L}{\partial b}$.

Calculating the gradients: A diagram



Calculating the gradients

From figure, we can see that the derivative product we ultimately want to compute is:

$$\frac{\partial \Lambda}{\partial P}(P,Y) \times \frac{\partial \alpha}{\partial N}(N,B) \times \frac{\partial \nu}{\partial W}(X,W)$$

First up: $\frac{\partial \Lambda}{\partial P}(P,Y)$

$\because \Lambda(P,Y) = (Y-P)^2$ for each element in $Y \in P$

$$\frac{\partial \Lambda}{\partial P}(P,Y) = 2(Y-P)\cdot(-1) \Rightarrow -2(Y-P)$$

code :- $\boxed{\texttt{dLdP = -2 * (Y-P)}}$

Net up: $\frac{\partial \alpha}{\partial N}(N,B)$, matrices, but α is just "addition"

with same logic as in chapter 1:
increasing any element of N by one unit will increase $P = \alpha(N,B) = N+B$ by one unit

Thus, $\frac{\partial \alpha}{\partial N}(N,B)$ is just matris +1ts, of the same shape as N.

code :- $\boxed{\texttt{dPdN = np.ones\_like(N)}}$

Finally : $\frac{\partial \nu}{\partial W}(X,W)$ from last chapter,

when computing derivatives of nested functions where one of the constituent functions is a matrix multiplication, we act as if:

$$\frac{\partial v}{\partial W}(X, W) = X^T.$$

Code: $\boxed{\text{dNdW = np.transpose(X, (1,0))}}$

```
dNdW = np.transpose(X, (1,0))
dPdB = np.ones_like(weights['B'])
```

Same as dPdN

Calculating the Gradients: The (Full) code

Goal: take all computed & inputed into the forward Pass

£ X, W, N, B, P and y

and compute:

$$\frac{\partial \Lambda}{\partial W} \;\&\; \frac{\partial \Lambda}{\partial B}$$

'W' & 'B' as inputs in a Dict. called "weights"

rest in a Dict. Forward_info:

returns loss_gradients, dict. containing weights & Bias. 'W', 'B' iterate over

Using These gradients to train the Model:

1. Select a batch of data
2. Run the forward pass of the model
3. Run the backward pass of the model using the info computed on the forward pass.
4. Use the gradients computed on the backward pass to update the weights.

Shuffling the data ~~or~~ to ensure that it is fed through in a random order.

We run the train function for certain num of "epochs", or cycles through the entire training dataset,

parameters :— weights, bias
hyper parameters ;- learning rate, epochs, batch_size,

Prediction :

We simply use the weights returned earlier from the train function & write :

code;- preds = predict (X_test, weights).

we just do matrix multiplication of X_test & weights['W'] & add [weights['B']] metrics

~~As~~ RMSE is particularly common metric. since it is on a some scale as the target. If we divide this number by mean of the target, we can get a measure of how far off a prediction is, on average, from it's actual value.

We calculate the mean of y-test: 22.0776

$$\frac{RMSE}{mean} = \frac{5.0508}{22.0776} = 22.9\% \text{ on average.}$$

Analyzing the most Important feature.

The reason why we do this analyze specific to Linear Regression is that we can interpret the absolute values of the coefficients as corresponding to the importance of the different features to the model; larger coefficient means the feature is more important.

Neural Networks from Scratch:

Step 1: A bunch of Linear Regressions

X = [batch_size, num_features] ⇒ [batch_size, 1]
W [num_features, 1]

matrix multiply

To do multiple regression's, we'll simply multiply our input by a weight matrix with dimensions

[num_features, num_outputs]

resulting in an output of dimensions [batch_size, num_outputs]; now, for each observation, we have num_outputs different weighted sums of the original features.

↳ We should think of each of them as "learned features" - a combination of the original features, that, once the network is trained, will represent to learn combina-tions of features that help it accurately predict.

Step 2: Non linear function:

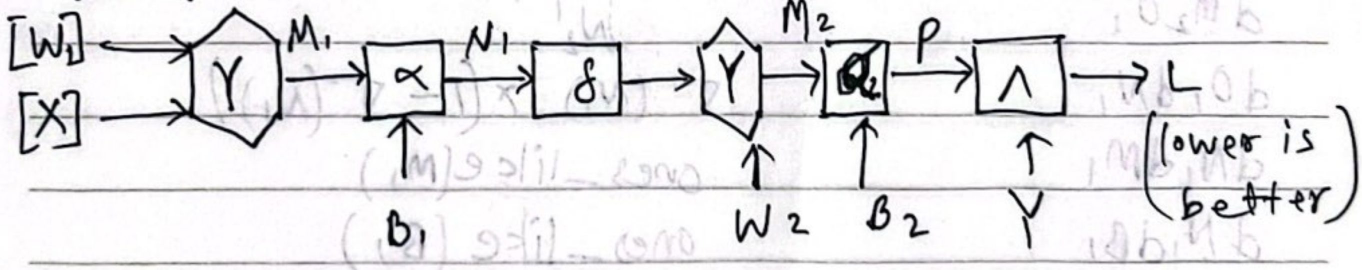
We ~~well~~ will sigmoid function to learn the non-linear pattern in the features. Also, to "preserve" information about the numbers th-at were fed in.

$$\frac{\partial \sigma}{\partial u}(x) = \sigma(x) \times (1 - \sigma(x))$$

Step 3: Another Linear Regression

we will feed the resulting elements from step 2 ~~through~~ containing values betw 0 and 1 into a regular linear regression and then we calculate MSE.

Diagram:



$W_1\ W_2 \rightarrow$ weights matrix

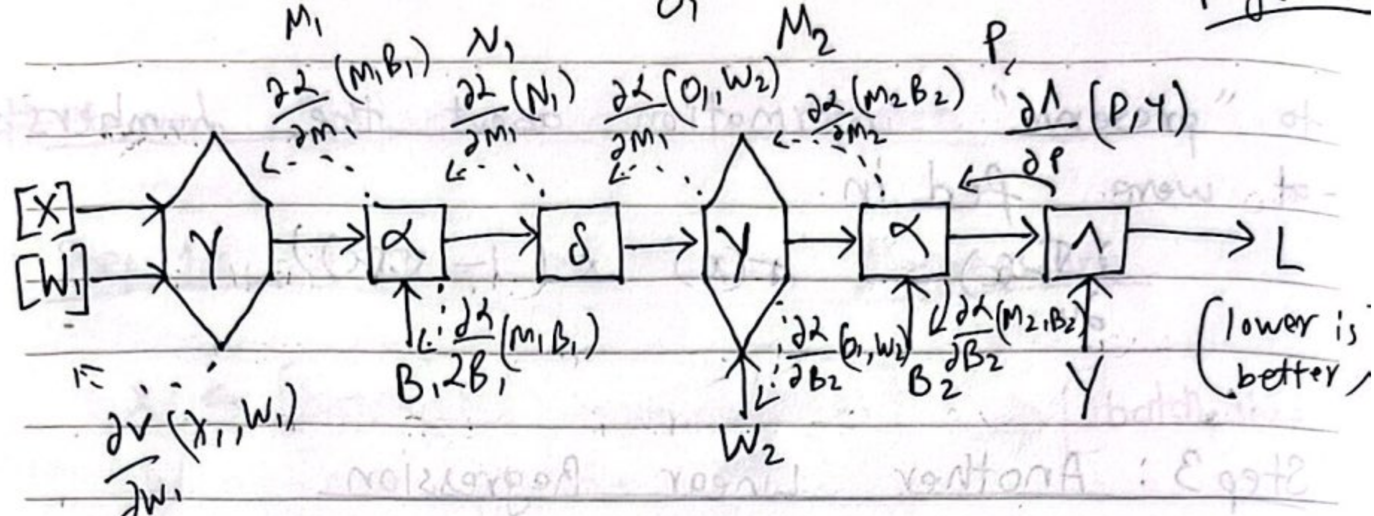
$B_1\ B_2 \rightarrow$ biases ~~ma~~

P — final prediction

Neural Network - Backward Pass

(1) Compute the derivative of each operation and evaluate it at its input.

(2) Multiply the results together.



Derivative rep.	Pseudo code : formula.
dLdP	$-(Y - P)$
dPdM$_2$	np.ones_like(M_2)
dPdB$_2$	np.ones_like(B_2)
dM$_2$dW$_2$	O_1^T
dM$_2$O$_1$	W_2^T
dO$_1$dN$_1$	$\sigma(N_1) \times (1 - \sigma(N_1))$
dN$_1$dM$_1$	ones_like(M_1)
dN$_1$dB$_1$	ones_like(B_1)
dM$_1$dW$_1$	X^T

From chain Rule we multiply to find:

dLdW$_2$ = dLdP × dPdM$_2$ × dM$_2$dW$_2$ ~~× dPdB$_2$ × dM$_2$dW$_2$~~

dLdB$_2$ = dLdP × dPdB$_2$ sum ~~× dM$_2$dB$_2$~~

dLdB$_1$ = dLdP × dPdM$_2$ × dM$_2$dO$_1$ × dO$_1$dN$_1$ × dN$_1$dB

dLdW$_1$ = dLdP × dPdM$_2$ × dM$_2$dO$_1$ × dO$_1$dN$_1$ × dN$_1$dM$_1$ × dM$_1$dW

dLdB1 = dLdN1 × dN$_1$B$_1$ .sum(axis=0)

Since we added 2-bias, we need to sum along axis=0 while computing derivative of output of Fully connected layer with respect to bias

Neural Network perform better than Linear Regression

- the NN is able to learn nonlinear relationships between the features and the target
- learn the relationships between combinations of features and the target.