-> Log loss -> When ~~Loss~~ CE is small -> high P

Error function: $-\frac{1}{m} \cdot \sum_{i=1}^{m} (1-y_i)(\ln(1-\hat{y}_i)) + y_i \cdot \ln(\hat{y}_i)$

-> Sigmoid is continious

- One hot encoding

Multi Class CE: $-\frac{1}{m} \sum_{i=1}^{m} \sum_{j=1}^{n} y_{ij} \cdot \ln(\hat{y}_{ij})$

Gradient Descent

$$E(W,b) = -\frac{1}{m} \sum (1-y_i) \ln(1-\sigma(W x^{(i)} + b)) + y_i \ln(\sigma(W x^{(i)} + b))$$

α has to be reasonable to not ended up in local min.

$\hat{y} = \sigma(Wx+b)$ <- bad

$\hat{y} = \sigma(w_1 x_1 + \dots + w_n x_n + b)$

$\nabla E = (\delta E/\delta w_1, \dots, \delta E/\delta w_n, \delta E/\delta b)$

$\alpha = 0.1$

$w_i' = w_i - \alpha \, \delta E/\delta w_i$

$b = b - \alpha \, \delta E/\delta b$

$\hat{y} = \sigma(W' x + b')$ <- better

$$\sigma'(x) = \frac{e^{-x}}{(1+e^{-x})^2} = \frac{1}{1+e^{-x}} \cdot \frac{e^{-x}}{1+e^{-x}} = \sigma(x) \cdot (1-\sigma(x))$$

$\frac{\delta}{\delta b} E = -(y - \hat{y})$ => closer the label to the pred, smaller the gradient

Sigmoid $\frac{1}{1+e^{-x}}$

$E = (y - \hat{y})^2$ Error for one pred.

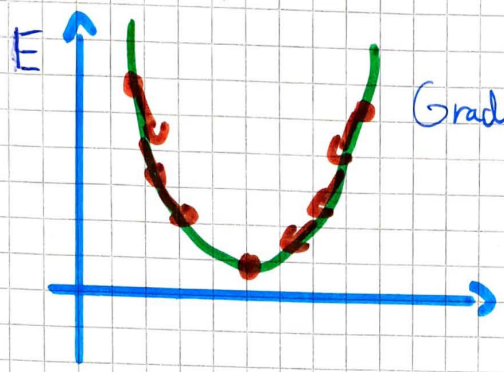
Total Error:

$$E = \frac{1}{2} \sum (y^\mu - \hat{y}^\mu)^2$$

$$\hat{y}^\mu = f\left(\sum_i w_i x_i^\mu\right)$$

$$-f\left(\sum_i w_i x_i^\mu\right))^2$$

→ SSE Sum Squared Error



Gradient Descent

$w_i = w_i + \Delta w_i$

$\Delta w_i \propto -\frac{\delta E}{\delta w_i}$ → The gradient

$$\frac{\delta E}{\delta w_i} = \frac{\delta}{\delta w_i} \frac{1}{2} (y - \hat{y})^2$$

$$= (y - \hat{y}) \cdot \frac{\delta}{\delta w_i} \cdot (y - \hat{y}) = -(y - \hat{y}) f'(h) x_i$$

Derivative of activation fct.

$\hat{y} = f(h)$ $h = \sum w_i x_i$

error $= (y - \hat{y})$

$\delta = (y - \hat{y}) f'(h)$ error term = error · output · (1 - output)

new $w_i = w_i + \eta \delta x_i$ $\Delta w = \alpha \cdot \text{error} \cdot x_i$

$\eta \delta x_i$ = Δw_i ↑ delta

Multiple Outputs:

$\delta_j = (y_j - \hat{y}_j) \cdot f'(h_j)$

$\Delta w_{ij} = \eta \delta_j x_i$

MSE

$$E = \frac{1}{2m} \sum_{\mu} (y^{\mu} - \hat{y}^{\mu})^2$$

Algo:

Set $\Delta w_i = 0$

for each record in training data:

$$\hat{y} = f(\Sigma_i w_i x_i)$$

$$\delta = (y - \hat{y}) \cdot f'(\Sigma_i w_i x_i)$$

$$\Delta w_i = \Delta w_i + \delta x_i$$

- update weights $w_i = w_i + \eta \Delta w_i / m$ (η: lr)
- repeat for e epochs

↳ Can be updated each step instead of each epoch

→ recommended to use normal distribution:

weights = np.random.normal(scale = $1/\sqrt{n}$, size =)

MLP

hidden_layer_in = np.dot(X, weights_input_to_hidden)

hidden_layer_out = sigmoid(hidden_layer_in)

output_layer_in = np.dot(hidden_layer_out, weights_hidden_to_output)

output_layer_out = sigmoid(output_layer_in)

Backprop

1. MLP code
2. error = target - output

output_error_term = error * output * (1 - output)

hidden_error_term = np.dot(output_error_term, weights_hidden_output) * hidden_layer_output * (1 - hidden_layer_output)

delta_w_h_o = α * output_error_term * hidden_layer_output

delta_w_i_h = α * hidden_error_term * x[:, None]

Complete Backprop:

(Loop Start)

hidden_input = X * del_w_input_hidden*

hidden_output = σ(hidden_input)

output = σ(hidden_output * w_hidden_output))

error = y - output

output_error_t = error * output * (1 - output)

hidden_error = output_error_term * w_hidden_output

hidden_error_term = hidden_error_term * hidden_output * (1 - hidden_output)

del_w_hidden_output += output_error_t * hidden_output

del_w_hidden_input += hidden_error_term * x[:, None]

(Weights_input_hidden = α * del_w_input_hidden / n_records

weights_hidden_output += α * del_w_hidden_output / n_records

Training

→ Train / Test set

→ simple model is better !

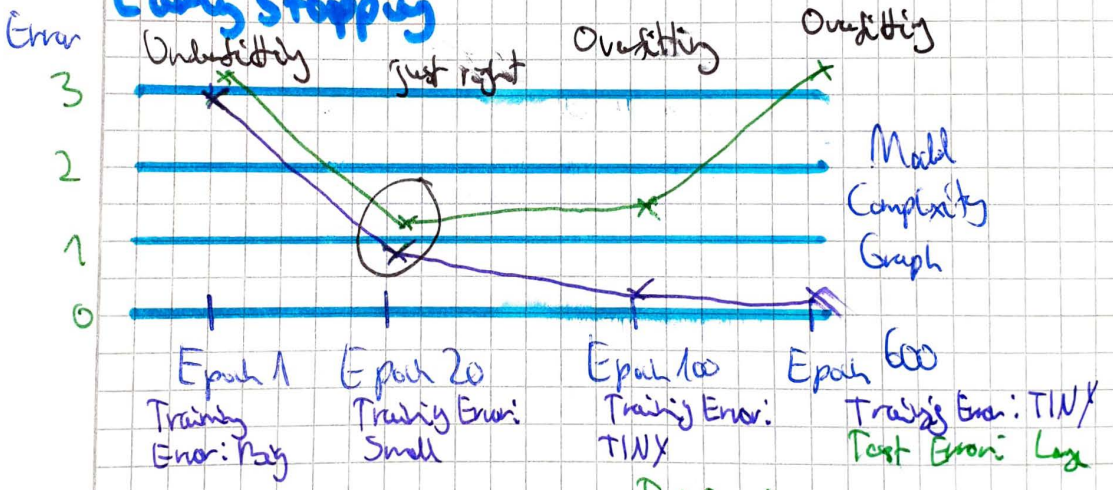
Overfitting

- Too complex, too specific — **High Variance**
- error due to variance
- instead of studying to memorize

Underfitting

- error due bias — **High Bias**
- Oversimplified

Early stopping



"The whole problem of bad model are certain of themselves and good ones are so full of doubts."

Large coefficients → ~~Large~~ Overfitting

Solution: Regularization — L_1 Reg → good for feature selection

Add $\lambda (|w_1| + \dots + |w_n|)$ or

$\lambda (w_1^2 + \dots + w_n^2)$ L_2 Reg. → good for training models

Weights are large we penalize

Dropout

-> turn parts off (Nodes) so other Nodes are trained better
-> Different Nodes are Dropped
-> 0.5 is recommended by researches, some nodes may never turn off though which is fine
-> best after layes w/ high # of params (not pooling!!)

Gradient Descent

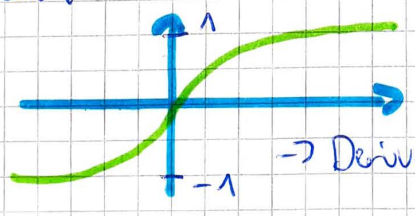
-> no local minimum, don't want to get stuck
-> Random restart to find global minimum

Vanishing Gradient Problem:

-> very tiny steps especially w/ Sigmoid -> Backprop decreases that
-> New activation Function:

tanh

$$\frac{e^x - e^{-x}}{e^x + e^{-x}}$$



-> Derivatives are larger

-> Final Output mostly Sigmoid

Batch GD

→ better than stochastic
→ quicker
→ small subsets of data
→ Data is split into badges:
Run points from one Badge and correct weights based on error. → small steps less accurate than large (stochastic), but almost there

Learning Rate Decay

LR too big:
may miss minimum, fast

LR too small:
Very ~~small~~ slow, get to minimum

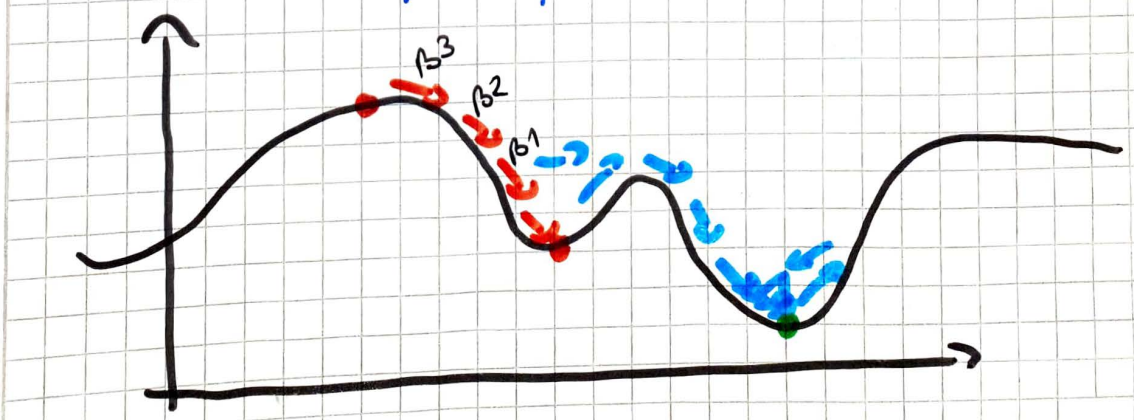
→ if error too big, decrease LR

Momentum

→ constant β: between 0 and 1

Step → average of previous steps → weighted (last steps more)

$Step(n) \rightarrow Step(n) + \beta Step(n-1) + \beta^2 Step(n-2) + \dots$



Sentiment Classification

(9)

- transform text to numeric problem so we can formulate correlations
1. curating dataset -> predictive theory for correlation
2. validate -> transform to output
3. iterate several times (increase correlation)

X: what we know (label) Y: what we want to know (comment)

-> Predictive Theory

-> which words are correlating w/ sentiment

2) word2index = {}

for i, word in enumerate(vocab):

word2index[word] = i

input layer hold amount = num of words in review (set)

for word in review.split(" "):

layer_0[0][word2index[word]] += 1

Good Initialization Method

$y = 1/\sqrt{n}$ n = number of inputs in the layer

is close to 0, but not too small

10

Neural Noise

-> Weighting interplays

-> many words are filler words => noise as these are most frequent

-> Remove noise from Input data

-> Binary representation

-> Set value of it to 1 if word exists
↑
word2index[word]

-> Which representations are relevant, which noise affects it most?

Solve inefficiencies

-> There is overhead

-> slow to train

-> what's stuff we can shave out?

-> e.g. Layer 0 is above 70000 weights
↳ Large multiplication w/ 0s doesn't change anything => shave it out

-> use part of the matrix that are not 0 to multiply

-> do not multiply by 1, it doesn't change anything

1) -> pre process training reviews so we can directly handle indices w/ non-zero inputs

2) Only use values that are non zero and update weight accordingly: for ind in review:
self.weights_0_1[index] -= ...

Pytorch tools

11

Autograd => calculates all the gradients and keeps track of them

.backward() for Backward pass through operations that created z

Optimizer

-> use SGD

->

General Process ~~torch.import~~ import NN, Optim
import torch.nn.functional as F

- make forward pass
- use output to calculate loss (loss)
- perform backward pass with .backward()
- take a step w/ the optimizer to update the weights

-> need to optimizer.zero_grad() when we do multiple passes to zero the gradient after one pass

-> Softmax gives a log probability
-> need an exponential for actual probabilities

Inference & Validation

- reduce Overfit by using Dropout
- accuracy measure used $\frac{TP+FP}{\text{all examples}}$

.view(* obj.shape) changes shape to shape of obj.shape tens

- ps. topk(1, dim = 1) returns most likely class
  ↳ tuple top-K values & indices (class label)

→ save model constantly and use early stopping
→ Dropout needs to be off during testing

Save Models

import fc_model → just model

save state_dicts → contains weights, bias, ...
→ torch.save(model.state_dict(), 'checkpoint.pth')

→ torch.load to load state dict
model.load_state_dict() to load dict in to model

Attention: model needs to be built just like it was when we trained →
checkpoint dict with all info
→ load checkpoint

Load Image Data

-> datasets.ImageFolder('path', transform = transforms)

root/class1/img1
root/class2/img2

-> transforms.Compose(

-> DataLoader to generate data in batches
-> iter(dataloader)

Augmentation -> from torchvision import transforms
-> Rotate, Scale, Change orientations, brightness
-> transforms.Compose([...])

Transfer learning

-> Pretrained Networks (ImageNet in this case)
import models with torchvision (also datasets)

-> train w/ Cuda GPU

Code:

device = torch.device("cuda:0" if
torch.cuda.is_available(), else "cpu")

-> freeze params so we don't backprop

-> only train last layers