

Deep learning - In detail

Mustafa Muhammad

24th October 2021

Input - Algorithm - Output credit-card-transactions - Algorithm - Fraud or Not?

Sum biases and weights, apply a non linear activation function like sigmoid and get the prediction.

Deep learning is good at solving complicated problems related to non linear solutions.

common eq

$$\hat{y} = x_1w_1 + x_2w_2$$

\hat{y} is the data, x is the data (features) and w is the weights, how important are the features to the outcome ?

The above function is good for solving linear problems, but what about non linear problems ?

$$\hat{y} = \sigma(x_1w_1 + x_2w_2)$$

σ is the activation or non linear function.

Simpler non linear functions work better than the complex ones. For example a log function.

It can also be called a neuron.

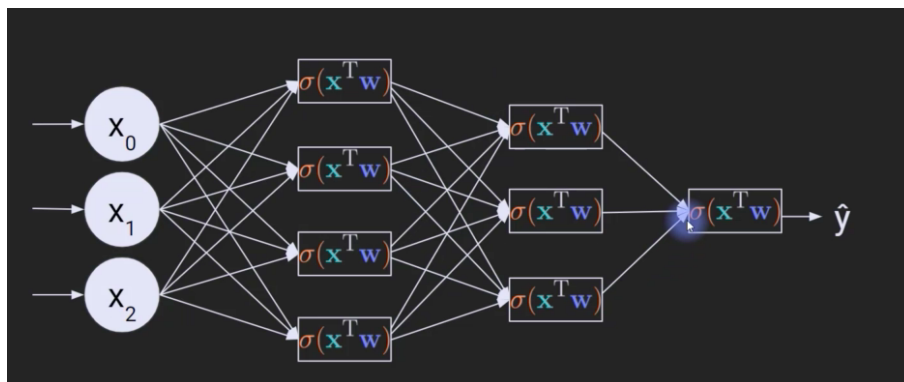


Figure 1: The simple equation gets repeated many times in a neural network

Steps

1. Forward propagation
2. Back propagation - Adjust weights based on feedback

Spectral theory - Breaking complex problems into smaller easier components.

Transpose vectors/matrices

```
#First row
```

```
nv = np.array([[1,2,3,4]])
```

```
nv = nv.T
```

```
#matrix
```

```
nM = np.array([[1,2,3,4],[5,6,7,8]])
```

```
nM = nM.T
```

```
#Using pytorch
```

```
tv = torch.tensor([[1,2,3,4]])
```

```
tvT = tv.T
```

```
tM = torch.tensor([[1,2,3],[4,5,6]])
```

```
tMT = tM.T
```

```
#Data types are different
```

Dot product $\alpha = a \cdot b = \langle a, b \rangle = a^T b = \sum_{i=1}^n a_i b_i$

$$\vec{p} = (x_1 \hat{i} + y_1 \hat{j} + z_1 \hat{k})$$

$$\vec{q} = (x_2 \hat{i} + y_2 \hat{j} + z_2 \hat{k})$$

$$\vec{p} \cdot \vec{q} = x_1 x_2 + y_1 y_2 + z_1 z_2$$

```
np.dot(nv1, nv2)
```

```
np.sum(nv1 * nv2)
```

```
torch.dot(tv1, tv2)
```

```
torch.sum(tv1 * tv2)
```

It is a single number that reflects the commonality between two objects.

Matrix Multiplication

```
np.matmul(A, B)
```

A@B If both are numpy arrays

Pytorch has same @ operators

multiplication and pytorch are friendly with multiplication.

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

Gives us values ranging between 0-1. Hence we can interpret these as probabilities.

Machine - Softmaxirizer

Takes in numerical quantities, applies the softmax function and the ouput is interpreted as the probability of the things occuring.

Larger numbers get larger softmax output.

sum over inputs : Any numerical value

sum over ouputs : Guaranteed to be 1.0

z = [1, 2, 3]

```
num = np.exp(z)
```

```
den = np.sum(np.exp(z))
```

```
sigma = num/den
```

```
softfun = nn.Softmax(dim=0)
```

```
sigmaT = softfun(torch.Tensor(z))
```

```
print(sigmaT)
```

Logarithms

Log is monotonic function of x. This is important because minimizing x is the same as minimizing log(x)

Log better distinguishes small and closely spaced numbers.

Entropy & cross entropy

Quantity that measures the amount of surprise that we have regarding the variable. Entropy in information theory: Everything is great and happy, and surprising things convey more information.

surprise : Amount of predictability or certainty. $H(x) = - \sum_{i=1}^n p(x_i) \log_2 p(x_i)$

x = datavalues p = probabilities

for coin flip n = 2

High entropy means that the dataset has a lot of variability. Low entropy means that most of the values of the dataset repeat (and therefore are redundant).

Entropy is nonlinear and makes no assumptions about the distribution.

Variance depends on the validity of the mean and therefore is appropriate for roughly normal data.

Cross entropy describes the relationship between two probability distributions.

$$-\sum p \log(q)$$

```
x = [0.25, 0.75]
```

```
H = 0
```

```
for p in x:
```

```
    H += -(p*np.log(p))
```

```
# Binary entropy — imp loss function in deep learning
```

```
H = -(p*np.log(p) + (1-p)*np.log(1-p))
```

```
# Cross entropy
```

```
p = [1, 0] # True label
```

```
q = [0.25, 0.75] # Model prediction
```

```
H = 0
```

```
for i in range(0, len(p)):
```

```
    H -= q[i]*np.log(p[i])
```

```
# pytorch
```

```
import torch.nn.functional as F
```

```
q_tensor = torch.Tensor(q)
```

```
p_tensor = torch.Tensor(p)
```

```
# Sensitive to the order of the inputs
```

```
F.binary_cross_entropy(p, q)
```

Min/Max and argmin/argmax min1, -2,1,2,3,4 = -1 max - so on

argmin & argmax find the location at which the smallest values occur. `v = np.array([1,2,3])`

`np.argmin(v)` and `np.argmax(v)`

Mean & Variance

Mean is the measure of central tendency.

Variance = measure of dispersion.

root of variance = standard deviation

Sampling variability

Different samples from the same population can have different values of the same measurement.

A single measurement may be an unreliable estimate of a population parameter.

Non random sampling can introduce systematic biases

Non representative sampling causes overfitting and limits generalizability.

Reproducible randomness via seeding

Global seed `np.random.seed(17)`

Newer seed mechanism local seed `np.random.RandomState(17)`

`torch.manual_seed(17)`

T-test

Determine data drawn from one distribution is better than data drawn from another distribution.

$$t = \frac{\bar{X} - \mu}{s/\sqrt{n}}$$

```
import scipy.stats as stats
```

```
n1 = 30
```

```
n2 = 40
```

```
mu1 = 1
```

```
mu2 = 2
```

```
data1 = mu1 + np.random.randn(n1)
```

```
data2 = mu2 + np.random.randn(n2)
```

```
t, p = stats.ttest_ind(data1, data2)
```

Derivates point in the direction of increases and decreases in a mathematical function. In deep learning, the goal (e.g classification) is represented by an error function. Thus, the best solution is the point with the smallest error.

The derivative tells us which way to move in that error landscape in order to find the optimal solution.

Gradient Descent

Fundamental algorithm in deep learning.

1. Guess a solution.
2. Compute the error.
3. Learn from mistakes and modify parameters.

Minimize the cost or loss function using GD.

Potential problems

It is guaranteed to go downhill, it can go wrong if the parameters are not set right for the particular landscape. Error landscapes are impossible to visualize in $\geq 2D$.

The success of deep learning, in spite of the "problems" with gradient descent, remains a mystery.

It is possible that there are many good solutions (many equally good local minima) . This interpretation is consistent with the huge diversity of weight configurations that produce similar model performance.

Another possibility is that there are extremely few local minima in high-dimensional space. This interpretation is consistent with the complexity and absurd dimensionality of DL models.

Saddle point, when a minima and maxima are in two different directions.

G.D will get trapped in a local minimum only if that point is a minimum in all dimensions.

Options ?

When model performance is good don't worry about local minima. One possible solution is to retrain the model many times using random weights and pick the model that does best.

Another solution is to add more dimensions, or make the model more complex to have fewer local minima.

```
localmin <- random point
learningRate <- 0.01
trainingEpochs <- 100

for i in range(trainingEpochs):
    grad <- computeGradient(localmin)
    localmin <- localmin - learningRate*grad

return localmin
```

Learning rate differs for every problem.

Vanishing gradient happens when the gradient gets extremely close to zero, effectively stopping the learning process.

Opposite of gradient descent is ascent. Used when to maximize a function.

```
localmin = localmin + learningRate*grad
```

Possible ways to proportionate the learning rate: 1, Training epoch: Good method often done in blocks. But unrelated to model performance or accuracy. This method is called "learning rate decay".

Initially the learning rate is high, but as the number of epochs increases, the learning rate starts to go down.

2, Derivative: Adaptive to the problem. Requires additional parameters and appropriate scaling. This method is incorporated into RMSprop and Adam optimizers.

```
steps: 1, calc grad 2, lr = learningRate * np.abs(grad) 3, localmin = localmin - lr*grad
```

3, Loss: Adaptive to the problem. Works only when loss is in range of [0,1].

4, Current local minimum value: Adaptive to the problem. Too many assumptions for this generally to be a good idea.

Vanishing gradient: when the function is constant (derivative is zero). problematic for deep networks since no learning.

Exploding gradient: Derivative so steep, we miss the minimum value, jumping past it. Weights change wildly - bad solutions.

How to minimize gradient problems

1. Use models with few hidden layers
2. Use activation functions that do not saturate (e.g ReLU)
3. Apply weight normalization
4. Pre-train networks using autoencoders
5. Use regularization techniques like batch normalization, dropout and weight decay
6. Use architectures like residual networks ("resnet")

The perceptron and ANN architecture

Linear: Addition and multiplication

NonLinear: Anything else

Linear models only solve linearly separable problems

Nonlinear models can solve more complex problems

Never use a linear model for nonlinear problem and vice versa.

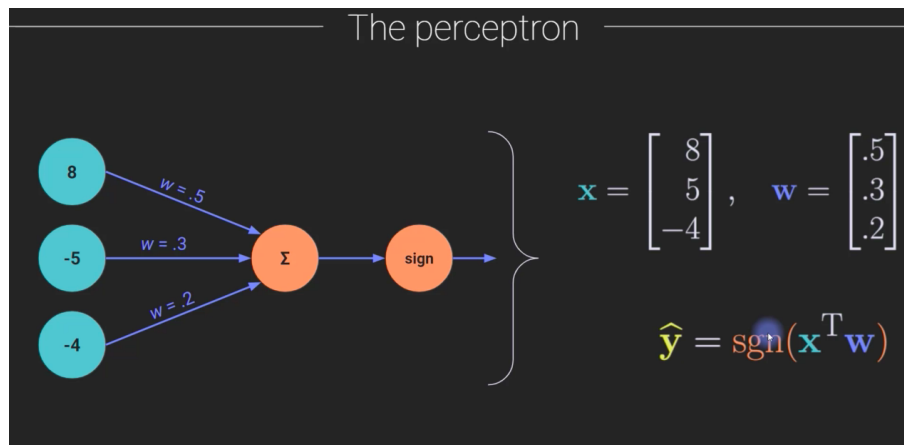


Figure 2: Non linear activation function applied on dot product of data and weights + bias (w_0)

bias - intercept - go off the origin

Feature space - A geometric representation of the data, where each feature of is an axis and each observation is a coordinate.

Separating hyperplane - A boundary that binarizes and categorizes data. It is used as a "decision boundary".

Forward propagation - Input data transformation to output. Deep learning - stringing multiple perceptrons together. $\hat{y} = \sigma(x_0 w_0 + \sum_{i=1}^m x_i w_i) = \sigma(w_0 + x^T w)$ Most used activation functions in deep learning.

1. Sigmoid
2. Hyperbolic Tangent
3. Rectified Linear Unit (ReLU)

Errors, loss & cost

y - target variable \hat{y} - prediction error = $\hat{y} - y$

Binarized error is easy to interpret, but less sensitive. Continuous error is more sensitive, but is signed.

We used continuous errors to teach the model and binarized ones to evaluate the model.

Errors are used to generate loss functions.

Most common loss functions

Mean-squared error (MSE)

Used for continuous data when the output is a numerical prediction. e.g height, house price, temperature. $L = \frac{1}{2}(\hat{y} - y)^2$ Squaring makes everything is positive. 0.5 is a convenience factor.

Cross-entropy (logistic)

Use for categorical data when the output is a probability. e.g, presence of disease, animal in picture, text sentiment. $L = -(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y}))$

Cost function - Average of the losses for many different samples. Compute losses for all data points and average them out. They are same with different names. $J = \frac{1}{n} \sum_{i=1}^n L(\hat{y}_i, y_i)$

We use loss as an optimization criterion in training. Goal: Find the set of weights that minimize the losses. $W = \arg_{\min}(J)$

Why train on cost and not loss?

Training on each sample is time consuming and may lead to overfitting.

But averaging over too many samples may decrease sensitivity.

A good solution is to train the model in "batches" of samples.

Back propagation - Adjust weights based on loss/cost

Each unit in feed forward step acts independently of each other.

Backprop is the same as gradient descent.

Updated local min is itself minus derivative scaled by learning rate $w = w - \eta \delta L$ eta is the learning rate, delta L is the derivative of the cost function.

ANN for regression

Simple regression means to predict one continuous variable from another.

$y_i = \beta_0 + \beta_1 x_i + \epsilon_i$ ith data value = intercept + coeff and iv + ith error (residual).

Pytorch Regression

```
import torch.nn as nn
import torch

x = torch.randn(30, 1)
y = x + torch.randn(30, 1)/2

model = nn.Sequential(
    nn.Linear(1, 1),
    nn.ReLU(),
    nn.Linear(1, 1)
)

loss = nn.MSELoss()
learning_rate = 0.05
optimizer = torch.optim.SGD(model.parameters(), learning_rate)

loss_arr = []

#Training loop template
for i in range(0, 500):
    predictions = model(x)

    # Compute loss
    actual_loss = loss(predictions, y)

    loss_arr.append(actual_loss.item())

    optimizer.zero_grad()

    # Backpropagation
    actual_loss.backward()

    optimizer.step()

preds = model(y)
```

Why use ML models, when DL works as well ?

Traditional ML models work good on small datasets, DL requires large amount of data. The ML models are better mathematically categorized, hence much easier to interpret as well.

For classification: Linear - ReLU - Linear - Sigmoid Loss function - Binary Cross Entropy

Sigmoid - Maps our data input between 0 and 1. Cut the data into two bins, category 0 and category 1.

The restricted numerical range increases stability and accuracy.

```
ANNclassify = nn.Sequential(  
    nn.Linear(2, 1),  
    nn.ReLU(),  
    nn.Linear(1, 1),  
    nn.Sigmoid()  
)  
  
lossFunc = nn.BCELoss()  
  
optimizer = torch.optim.SGD(ANNclassify.parameters(), lr=learningRate)  
  
#After training  
  
predictions = ANNclassify(data)  
  
predlabels = predictions>0.5  
  
#Pytorch recommends using BCEWithLogistLoss() instead of BCELoss() since its  
more stable, when using BCEWithLogitsLoss(), we do not need to initialize sigmoid  
layer in our model. It is handled by the loss function itself.
```

Output features have to match input features of the next layer.

Difference between linear and non linear models.

Why multilayer linear models don't exist ?

This is because if there's no non linear activation function, all the layers collapse into one. (Formula for perceptron is just repeatedly applied).

Comparing the number of hidden units.

Pytorch class

```
class ANN(nn.Module):  
    def __init__(self):  
        super().__init__()  
        self.input = nn.Linear(2, 1)  
        self.output = nn.Linear(1, 1)
```

```
def forward(self, x):
    x = self.input(x)
    x = torch.nn.functional.relu(x)
    x = self.output(x)
    x = torch.sigmoid(x)
    return x
```

Are DL models understandable ?

The nonlinearities and interactions across hundreds of parameters (weights) means that we have no idea what each node is actually encoding.

Regularization

Penalizes "memorization" (overlearning examples)

Helps model to generalize to unseen examples

Changes the representations of learning (either more sparse or distributed, depending on regularization)

It can also increase or decrease training time.

Generally it tends to decrease training accuracy.

Works better for large models with many hidden layers.

Generally works better with sufficient data.

Three families of regularizers.

1. Modify the model (dropout)
2. Add a cost to the loss function (L1/L2).
3. Modify or add data (batch training, data augmentation).

Dropout - "Remove" nodes randomly during learning by forcing the output of a unit to equal 0.

L1/L2 regularization ("weight decay")

Add a cost to the loss function to prevent the weights from getting too large.

$$\text{Cost Function} = J = \frac{1}{n} \sum_{i=1}^n L(\hat{y}_i, y)$$

$$\text{Regularized Cost Function} = J = \frac{1}{n} \sum_{i=1}^n L(\hat{y}_i, y) + \textit{something}$$

Making sure weights stay in a range reasonably close to zero.

Data Augmentation

Add more data as slightly modified versions of existing data (usually just for images).

How to think about regularization?

1. Adds a cost to the complexity of the solution.
2. Forces the solution to be smooth.
3. Prevents the model from learning item specific details.

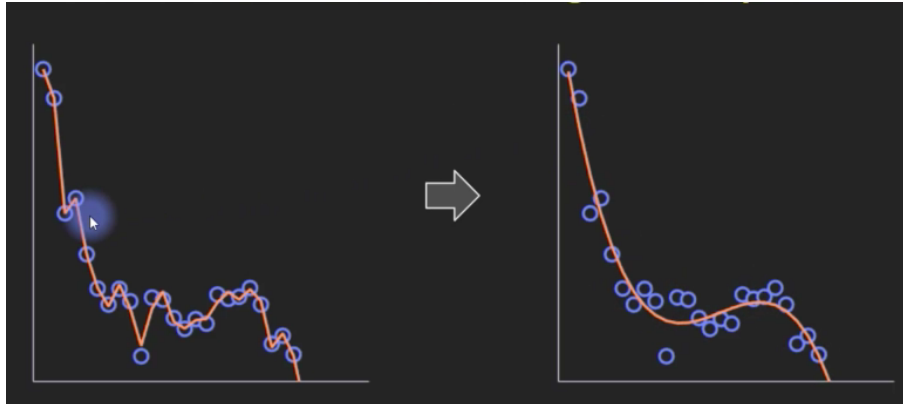


Figure 3: Making the go from overfitting the data to a much smoother solution.

Which regularizer to use ?

Usually the best method is problem or achitecture specific.

`train()` and `eval()` modes in pytorch

Gradients are computed only during backpropagation and not during evaluation.

Some regularization methods are applied only during training and not during evaluation.

Ergo: We need a way to deactivate gradient computations and regularization while evaluating model performance.

```
net.train() , net.eval() \& torch.no_grad()
```

Training mode, testing mode & Used in testing mode.

Regularization active, off & gradients not computed

Necessary for training regularization - on by default, Necessary for evaluation & Never necessary - makes large models evaluate faster.

```
training epoch loop:
    net.train()
```

```
    #Train the model
    batch loop:
```

```
    #Evaluate the model
```

```

net.eval()
with torch.no_grad():
    yHat = net(X)

```

Dropout Effects

1. Prevents a single node from learning too much.
2. Forces model to have distributed representations.
3. Makes the model less reliant on individual nodes, thus more stable.

Other observations

1. Generally requires more training, even though each epoch computes faster
2. Can decrease training accuracy, however increases model generalization
3. Usually works better on deep than shallow networks
4. Debate about applying to convolution layers
5. Works better with sufficient data, unnecessary with "enough" data.

```

import torch
import torch.nn as nn
import torch.nn.functional as F

```

```

prob = 0.25

```

```

dropout = nn.Dropout(prob)

```

```

x = torch.ones(10)
y = dropout(x)
print(y)

```

```

#dropout is turned off when evaluating the model
dropout.eval()
y = dropout(x)
print(y)

```

```

#Two methods of using dropout, sequential and functional
y = F.dropout(x, training=False)

```

```

class Model(nn.Module):
    def __init__(self, dropout_rate):
        super().__init__()

        self.input = nn.Linear(4, 12)
        self.hidden = nn.Linear(12, 12)
        self.output = nn.Linear(12, 3)

```

```

self.dr = dropout_rate

def forward(self, x):
    x = F.relu(self.input(x))
    x = F.dropout(x, p=self.dr, training=self.training)
    x = F.relu(self.hidden(x))
    x = F.dropout(x, p=self.dr, training=self.training)
    x = self.output(x)

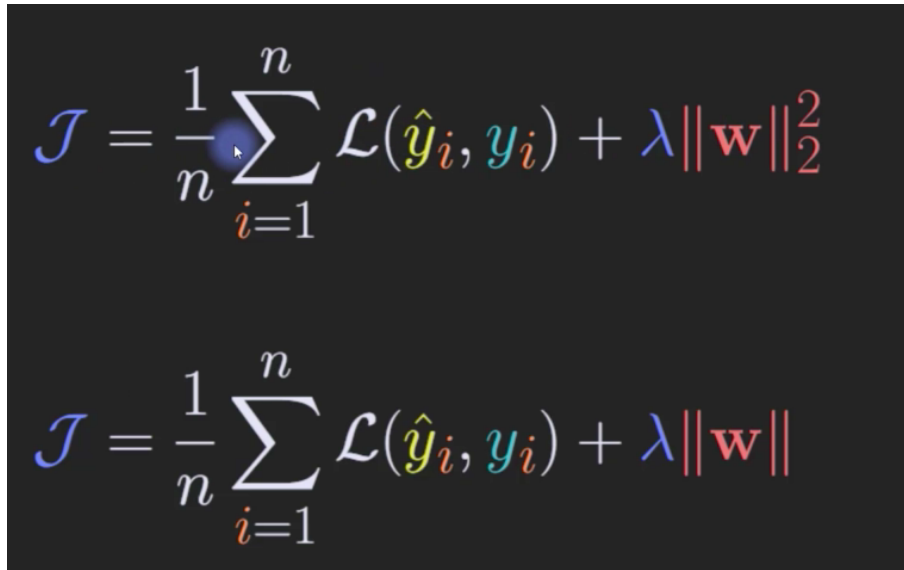
    return x

#Training loop
model = Model(0.2)
for epoch in range(num_epochs):
    model.train()
    # Forward step and backprop
    model.eval()
    # Model evaluation

```

L1 and L2 regularization Main goal = $W = \arg_w \min(J)$

$$J = \frac{1}{n} \sum_{i=1}^n L(\hat{y}_i, y_i) + \text{penalty}$$



$$\mathcal{J} = \frac{1}{n} \sum_{i=1}^n \mathcal{L}(\hat{y}_i, y_i) + \lambda \|\mathbf{w}\|_2^2$$

$$\mathcal{J} = \frac{1}{n} \sum_{i=1}^n \mathcal{L}(\hat{y}_i, y_i) + \lambda \|\mathbf{w}\|$$

Figure 4: Top: L2, Bottom: L1 regularization

magnitude of w - lower 2 norm - upper square

L2 regularization - ("ridge" or "weight decay") - Shrinks all weights, especially large weights.

L1 regularization - ("lasso") - Creates sparse weights by setting some to 0.

Model is pressured to reduce the weights when the penalty is large. When the weights are small the penalty is small.

The cost function is relatively large compared to the penalty term.

L1+L2 ("elastic net" regression)

Norm of weight matrix

Sample-specific (e.g positive bias on cancer diagnosis)

why does regularization reduce overfitting ?

1. Discourages complex and sample specific representations.
2. Prevents overfitting to training examples.
3. Large weights lead to instability (very different outputs for similar inputs).

When to use ?

In large, complex models with lots of weights (High risk for overfitting).

Use L1 when trying to understand the important encoding features (more common in regression than DL).

When training accuracy is much higher than validation accuracy.

L2 regularization in practice.

It is specified in the optimizer in Pytorch.

Default L2 is 0 and is a floating point number.

L2 = 0.01

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.005, weight_decay=L2)
```

L1 regularization in practice.

Pytorch does not provide a direct implementation for L1 regularization.

Activation functions dont have any parameters

Before training loop

n_weights = 0

```
for pname, weights in model.named_parameters():
```

```
    if 'bias' not in pname:
```

```
        n_weights = n_weights + weight.numel()
```

Inside the training loop – after computing the loss

```

L1_term = torch.tensor(0., requires_grad=True)

for pname, weights in model.named_parameters():
    if 'bias' not in pname:
        L1_term = L1_term + torch.sum(torch.abs(weights))

loss = loss + L1_lambda*L1_term/n_weights

# Continue on with backpropagation

```

Training in mini batches.

batch size = 1 is "Stochastic gradient descent"

Subset of the total data - we call that a mini batch - we run forward and backprop on that data.

How and why to train with mini batches ?

Batch size is often powers of 2, between 2 and 512.

Training in batches can decrease computation time because of vectorization. (matrix multiplication instead of for loops).

But batching can increase computation time for large batches and large data samples (e.g. images)

Batching is a form of regularization: it smoothes learning by averaging the loss over many samples, and thereby reducing overfitting.

If samples are highly similar, minibatch=1 can give faster training.