

# How does the NN "learn" the transformations ?

The matrix  $\mathbf{W}$  contains the "patterns" that serve to recognize the synthetic features created by each layer

- $\mathbf{W}_{(l),j}$  are the weights /pattern for feature  $\mathbf{y}_{(l),j}$

How are these patterns discovered ?

The answer is: exactly as we did in Classical Machine Learning

- Define a loss function that is parameterized by  $\mathbf{W}$ :

$$\mathcal{L} = L(\hat{\mathbf{y}}, \mathbf{y}; \mathbf{W})$$

- Per example loss  $\mathcal{L}^{(i)}$

- Average loss  $\mathcal{L} = \frac{1}{m} \sum_{i=1}^m \mathcal{L}^{(i)}$

- Our goal is to find  $\mathbf{W}^*$  the "best" set of weights

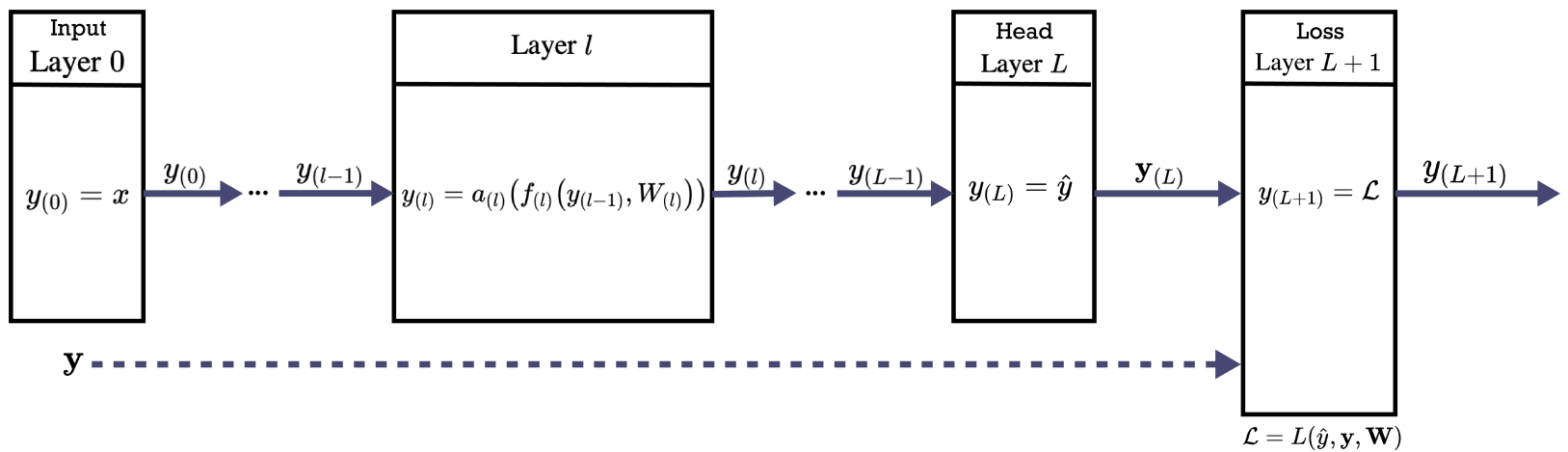
$$\mathbf{W}^* = \underset{\mathbf{W}}{\operatorname{argmin}} L(\hat{\mathbf{y}}, \mathbf{y}; \mathbf{W})$$

- Find  $\mathbf{W}^*$  using Gradient Descent !

Very much in spirit of the multi-layer architecture

- We add a new layer (L+1) to compute the loss  $\mathcal{L}$  !

## Additional Loss Layer (L+1)



# Gradient Descent review

Gradient Descent is an iterative method for finding the minimum of a function. <!--EdX:  
Omit this from EdX: can't refer to prior course

- See the [Gradient Descent lecture \(Gradient Descent.ipynb\)](#) in the Classical ML part of the course for more details -->

Let's review Gradient Descent using our current notation

- We start with an initial guess for  $\mathbf{W}$  and iteratively improve it.
- Compute the loss  $\mathcal{L}$  given the current  $\mathbf{W}$ 
  - Average loss of the  $m$  examples in the training examples
- Compute the gradient

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}}$$

- Update  $\mathbf{W}$  in the direction of the *negative* of the gradient
- Scaled by a learning rate  $\alpha$

$$\mathbf{W} = \mathbf{W} - \alpha * \frac{\partial \mathcal{L}}{\partial \mathbf{W}}$$

A unit change in  $\mathbf{W}$  increases  $\mathcal{L}$  by  $\frac{\partial \mathcal{L}}{\partial \mathbf{W}}$

- That's why there is a negative sign: we proceed in the direction *opposite* the one that increases  $\mathcal{L}$
- We move only a fraction  $\alpha \leq 1$  of the (negative) of the gradient
- To avoid the possibility of over-shooting the minimum

$\mathbf{W}$  is a multi-dimensional vector, not a scalar

- So the gradient is multi-dimensional
- We will formally discuss Matrix Gradients in a later lecture
  - For now: we compute the derivative with respect to each element of  $\mathbf{W}$  and arrange in a matrix



The loss  $\mathcal{L}$  is averaged over all  $m$  training examples.

This can be expensive to compute.

We can approximate  $\mathcal{L}$  by *sampling* from the  $m$  training examples

- Choose a *random subset* (of size  $m' \leq m$ ) of examples:  $I = \{i_1, \dots, i_{m'}\}$
- Approximate  $\mathcal{L}$  on  $I$

$$\mathcal{L} \approx \frac{1}{|I|} \sum_{i \in I} \mathcal{L}^{(i)}$$

## Minibatch gradient descent

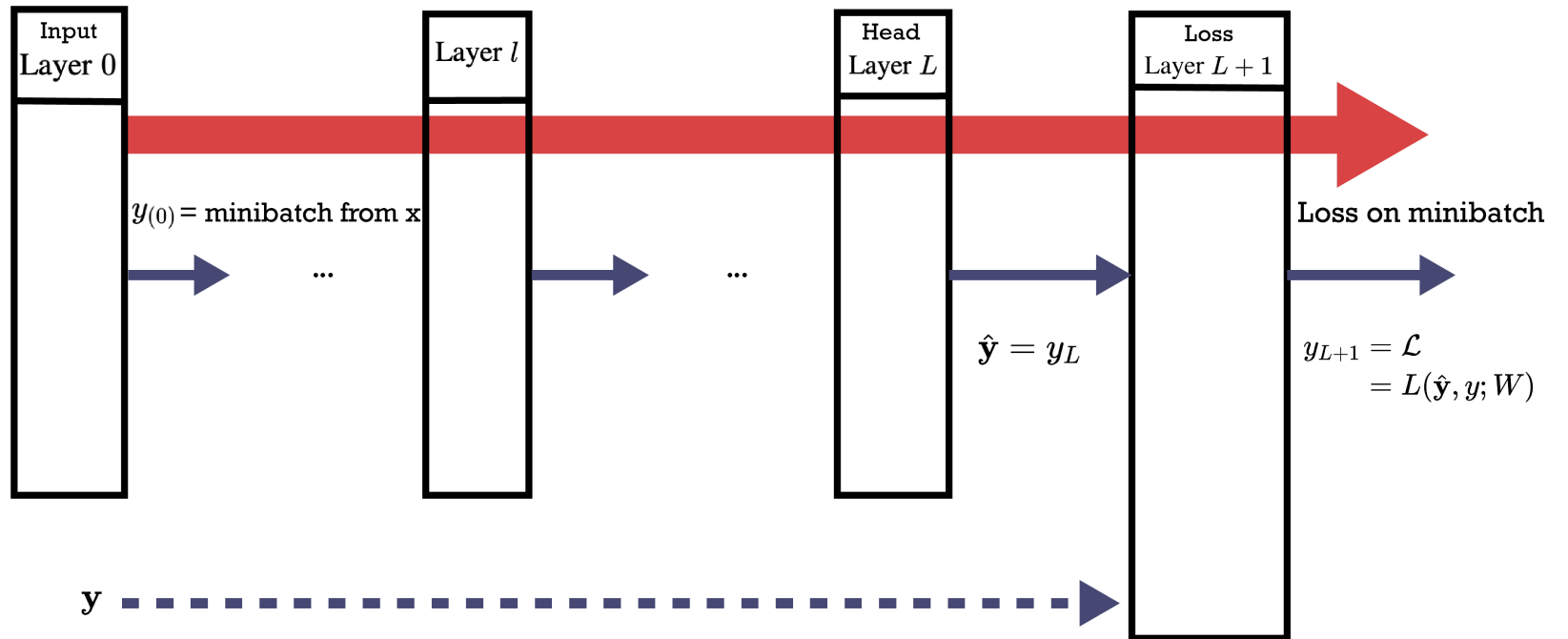
- Divides the  $m$  training examples
- Into  $b = m/m'$  disjoint batches of  $m' \leq m$  examples each

- Iterates over the batches
  - Approximate the loss on the current batch
  - Update  $\mathbf{W}$  according to

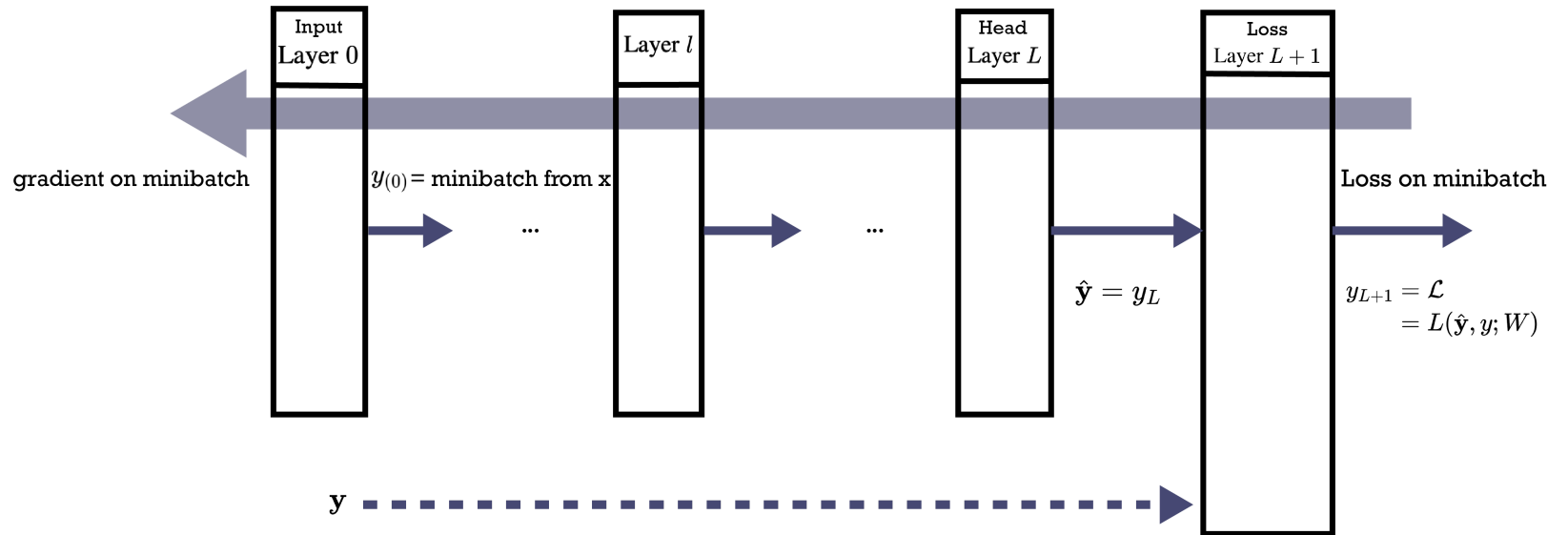
$$\mathbf{W} = \mathbf{W} - \alpha * \frac{\partial \mathcal{L}}{\partial \mathbf{W}}$$

- Repeat until all the batches have been processed

## Minibatch: Forward Pass From minibatch to Loss



## Minibatch: Backwards Pass From minibatch Loss to Gradient



Thus, Minibatch Gradient Descent

- Examines *all*  $m$  training examples
- In batches of size  $m' \leq m$
- Resulting in  $b = m/m'$  updates to  $\mathbf{W}$  for each complete pass through the  $m$  training examples

Minibatch Gradient Descent is faster than a single batch of size  $m$

- Update  $\mathbf{W}$   $b$  times, rather than once
- A complete pass through the  $b$  mini-batches is called an *epoch*

# The Training loop

A single epoch of Gradient Descent encounters all  $m$  examples and makes  $b$  updates

We may need additional epochs to continue to drive down the Loss.

This iterative process is called the *training loop*.

Here is some pseudo-code:



```
initialize(W) # Training loop to implement mini-batch SGD for epoch in range(n_epochs):` for X_batch,  
y_batch in next_batch(X_train, y_train, batch_size, shuffle=True): # Forward pass y = NN(X_batch) # Loss  
calculation loss = loss_fn(y, y_batch) # Backward pass grads = gradient(loss, W) # Update  $W = W - \text{grads} * \text{learning\_rate}$ 
```

It used to be the case that this fairly standard training loop was coded for each problem.

Just as `sklearn` wrapped common code into a high-level API

- We will use a toolkit that hides the training loop behind a high level API

# Scaling the inputs

Many times in this course we have pointed out that some models are *scale sensitive*.

Neural Networks are not *mathematically* sensitive but tend to be so *in practice*.

It is *highly recommended* to scale your data so their absolute values are around 1.0 or at least somewhat small.

Gradient Descent is the root of the problem:

- Two features on different scales can cause the optimizer to favor one over the other
- Activations can *saturate*
  - Output of dot product (Dense layer) is in the "flat\*" area of the activation
  - Zero derivative: no learning
- The Cost/Loss may be large in initial epochs when the target values are too different from the dot products
  - *Large* gradients: unstable learning
  - Weights are typically initialized to values less than 1.0, leading to small dot products

Remember: if you re-scale the inputs, you will need to invert the transformation when communicating the results.

In [4]: `print("Done")`

Done