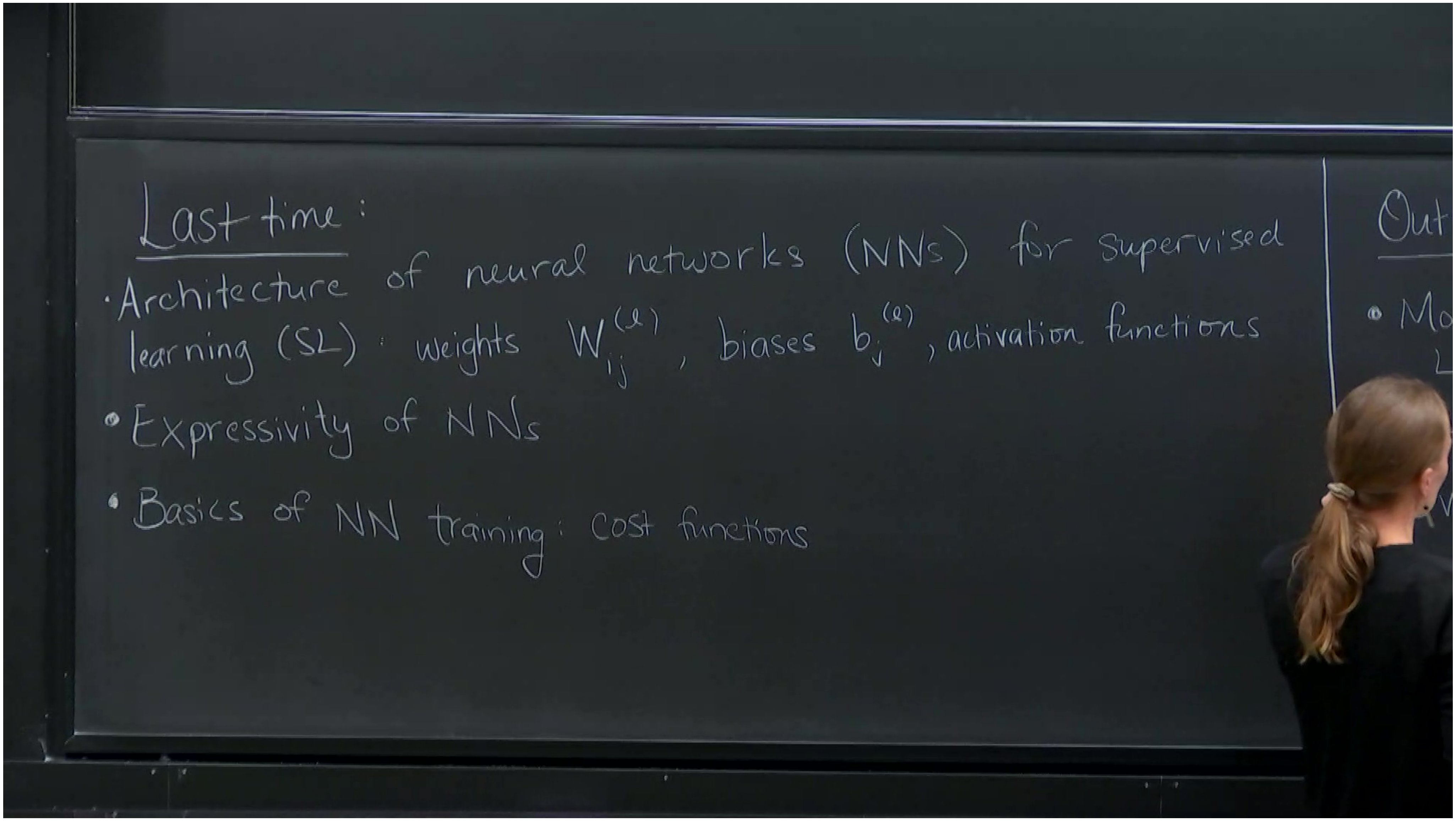
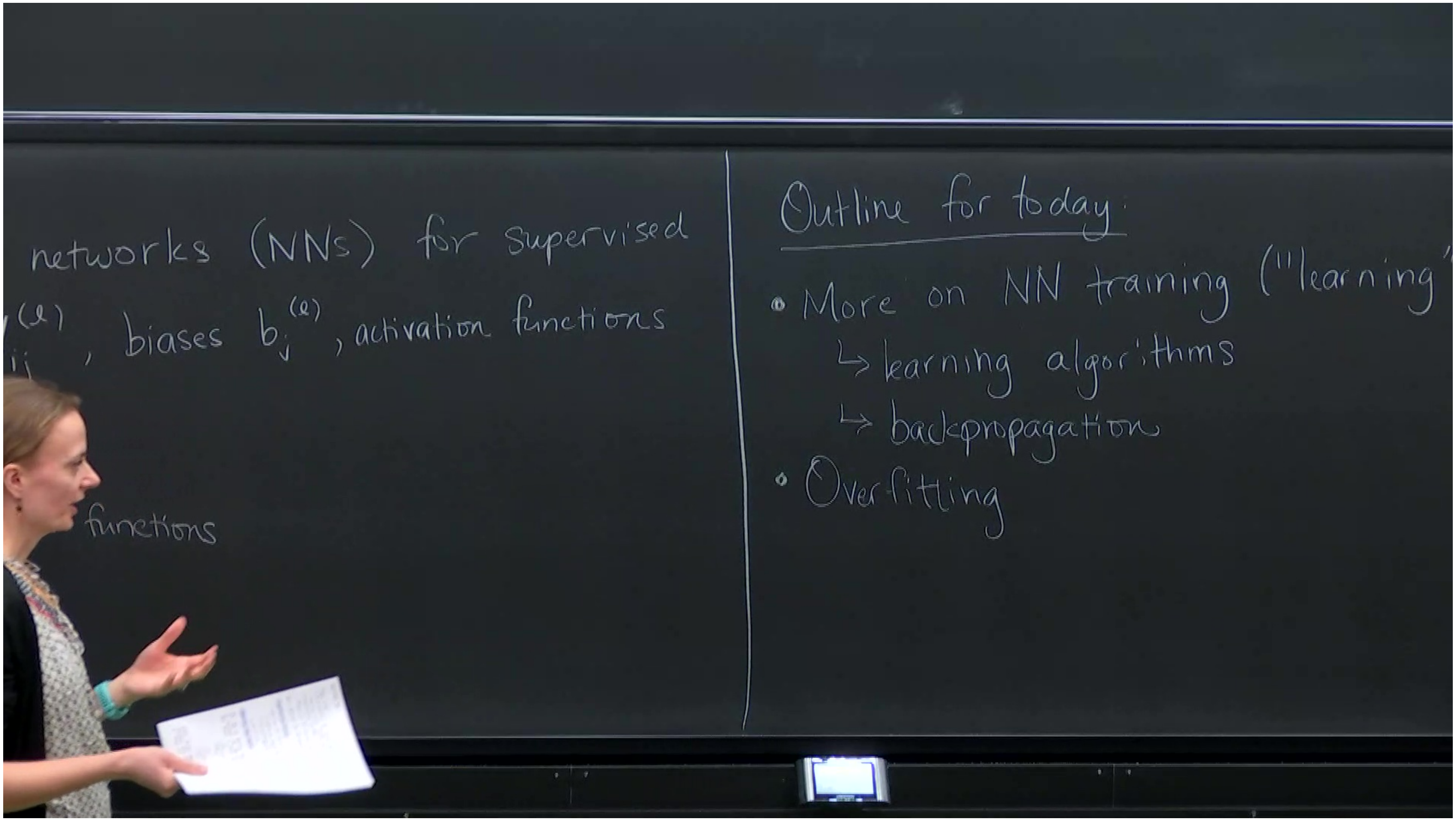


Last time:

- Architecture of neural networks (NNs) for supervised learning (SL): weights $W_{ij}^{(\ell)}$, biases $b_i^{(\ell)}$, activation functions
- Expressivity of NNs
- Basics of NN training: cost functions

Out

- Mo



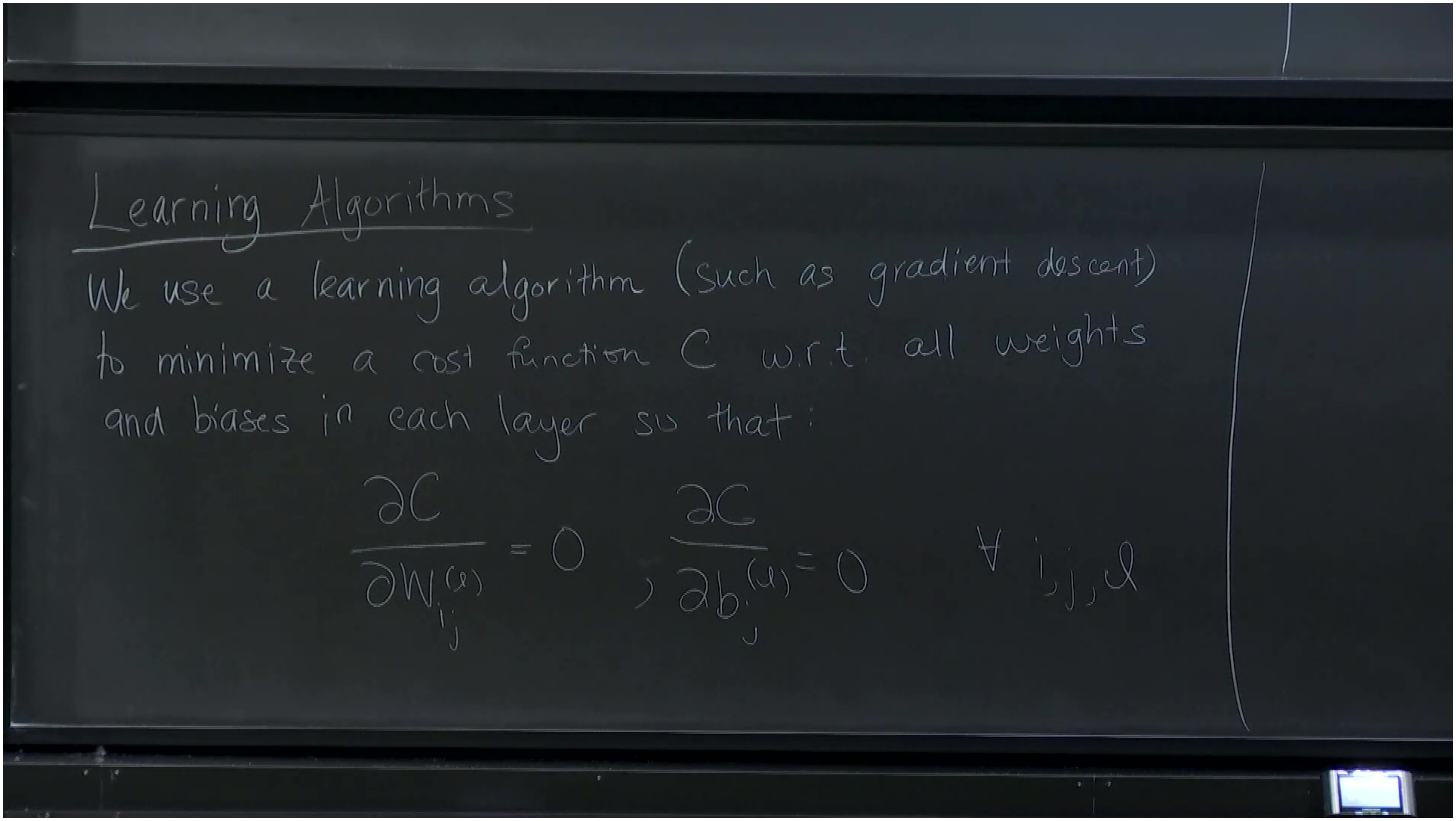
networks (NNs) for supervised

$w^{(\ell)}_{ij}$, biases $b^{(\ell)}_j$, activation functions

functions

Outline for today:

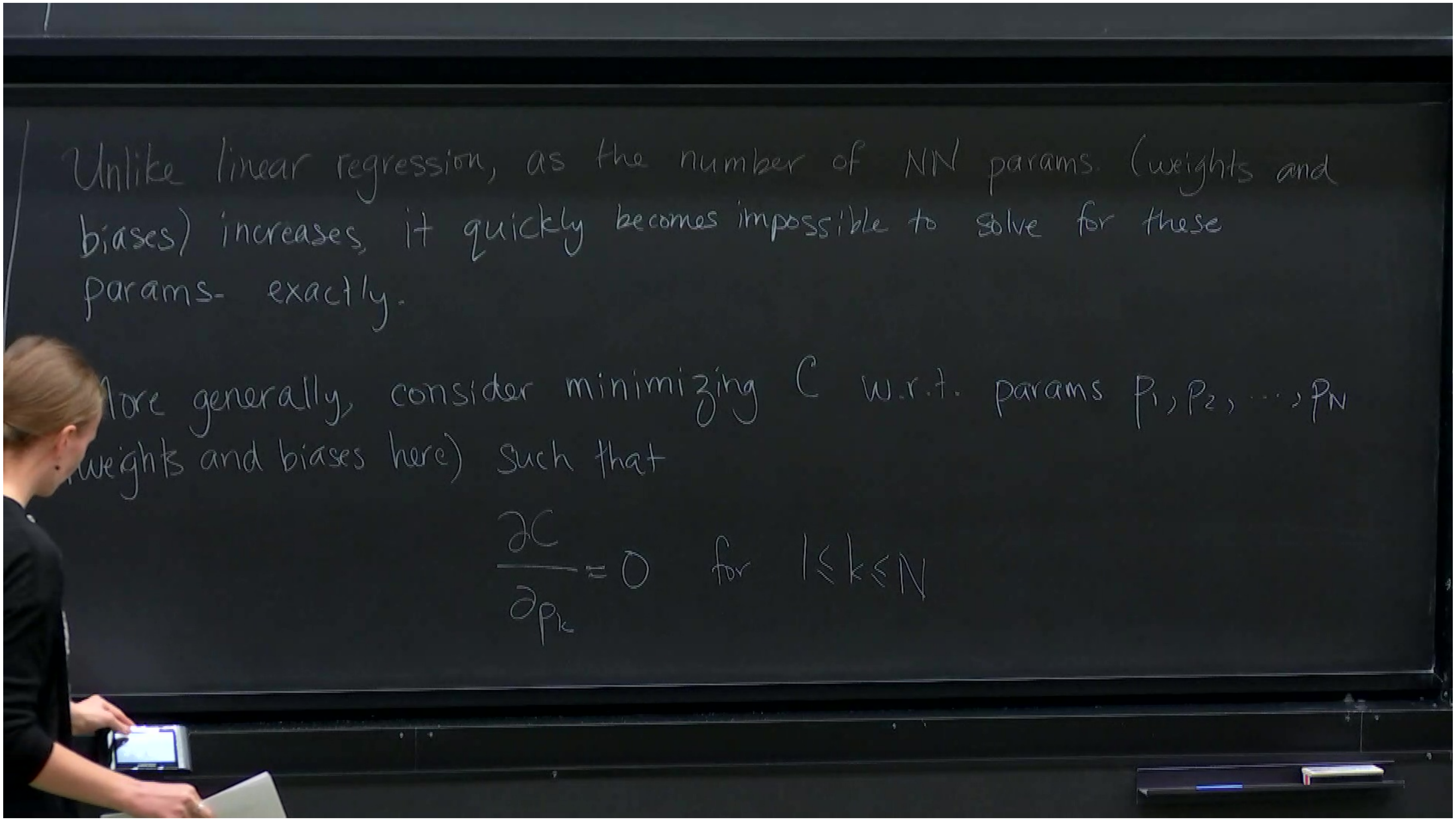
- More on NN training ("learning"
  - $\hookrightarrow$ learning algorithms
  - $\hookrightarrow$ backpropagation
- Overfitting



Learning Algorithms

We use a learning algorithm (such as gradient descent) to minimize a cost function C w.r.t. all weights and biases in each layer so that:

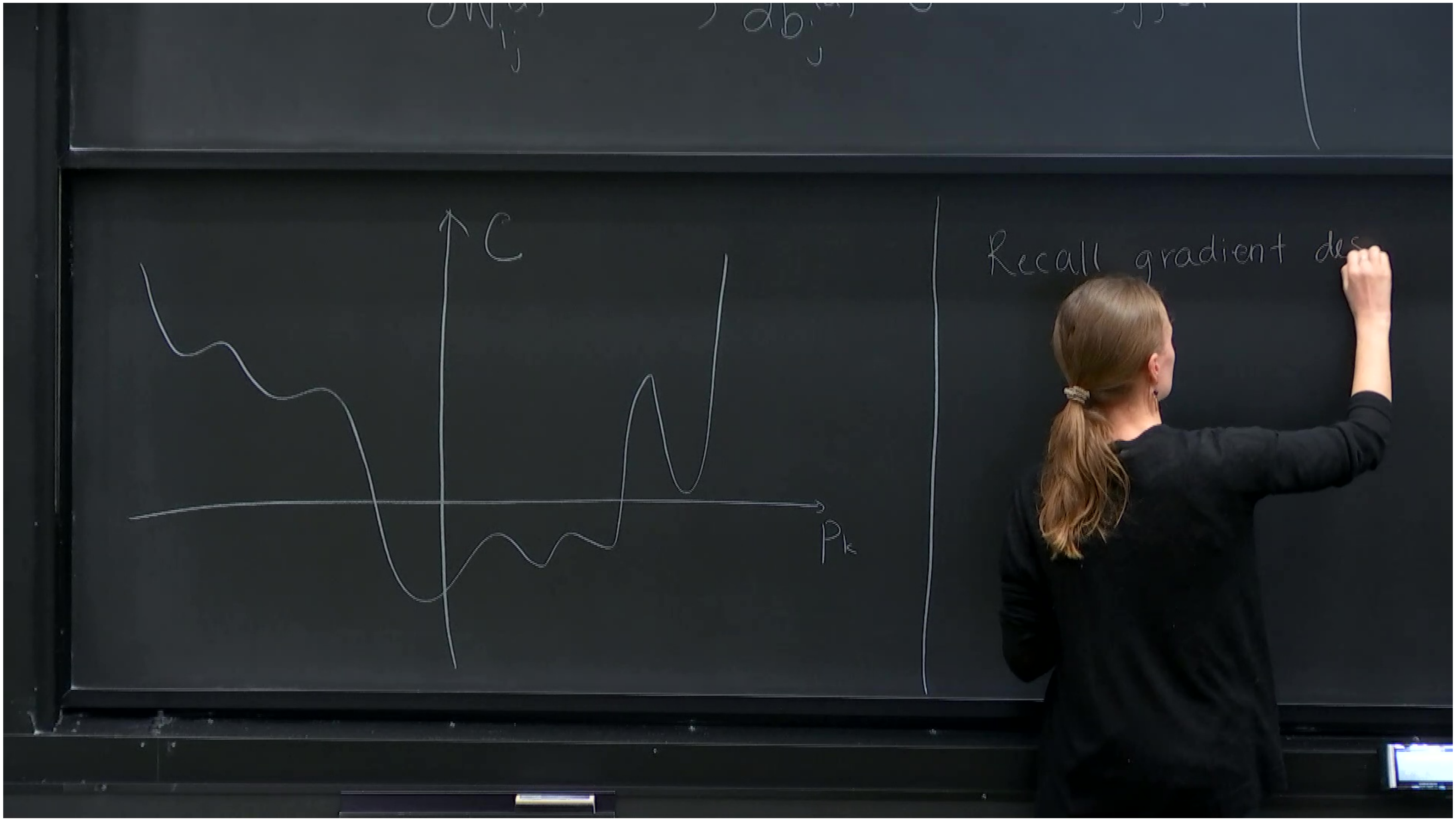
$$\frac{\partial C}{\partial w_{ij}^{(\ell)}} = 0 \,, \quad \frac{\partial C}{\partial b_j^{(\ell)}} = 0 \qquad \forall\ i, j, \ell$$



Unlike linear regression, as the number of NN params (weights and biases) increases, it quickly becomes impossible to solve for these params exactly.

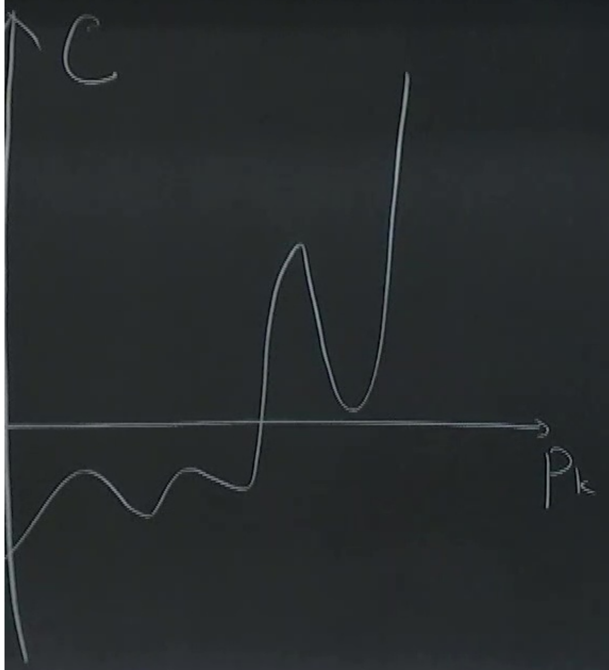
More generally, consider minimizing C w.r.t. params $p_1, p_2, \dots, p_N$ (weights and biases here) such that

$$\frac{\partial C}{\partial p_k} = 0 \quad \text{for } 1 \leq k \leq N$$



Recall gradient des

∂p_k

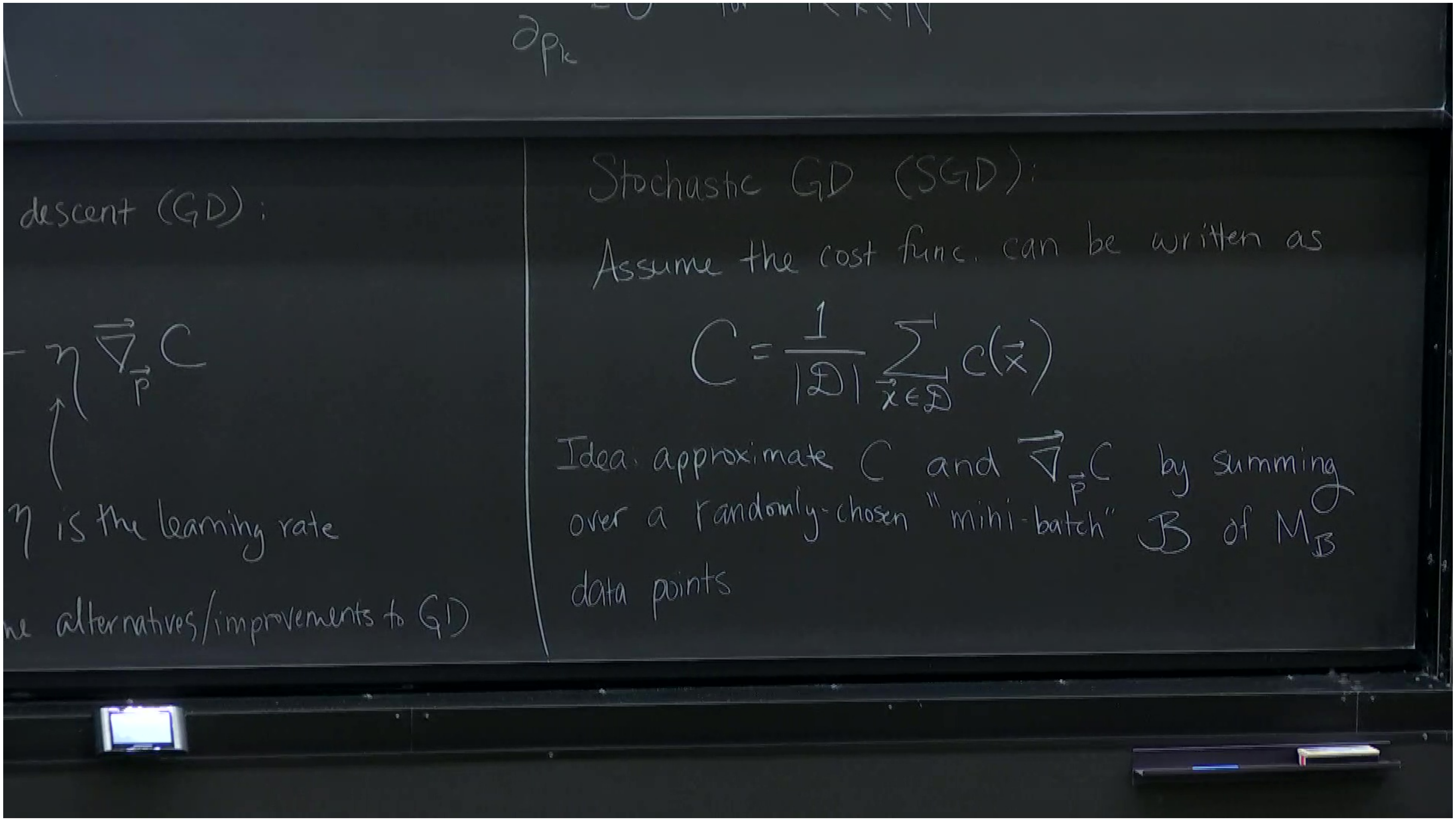


Recall gradient descent (GD):

$$\vec{p} \longrightarrow \vec{p} - \underset{\uparrow}{\eta}\, \vec{\nabla}_{\vec{p}} C$$

η is the learning rate

Let's now examine some alternatives/improvements to GD



∂p_k

descent (GD):

$- \eta \vec{\nabla}_{\vec{p}} C$

η is the learning rate

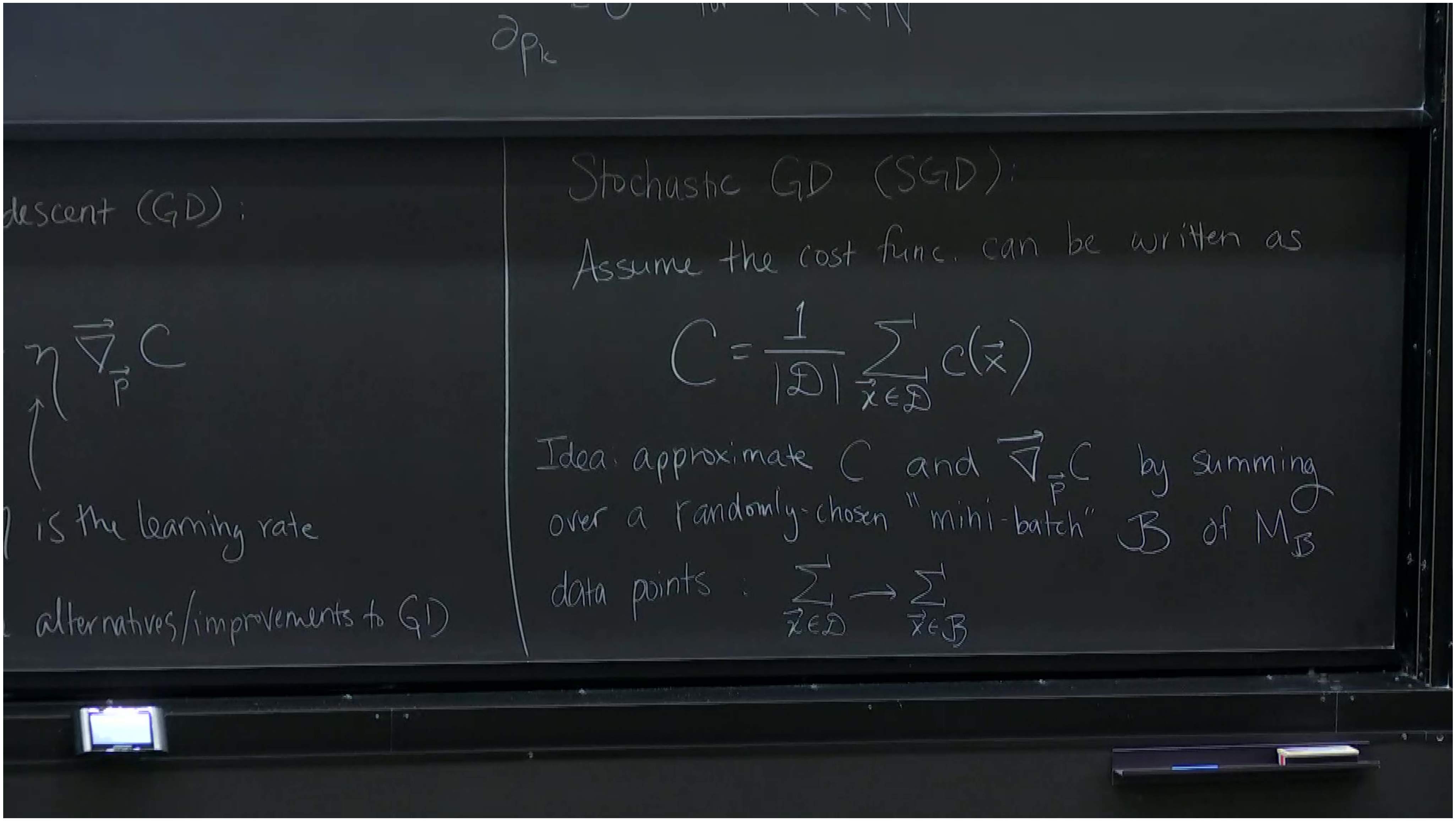
ne alternatives/improvements to GD

Stochastic GD (SGD):

Assume the cost func. can be written as

$$C = \frac{1}{|\mathcal{D}|} \sum_{\vec{x} \in \mathcal{D}} c(\vec{x})$$

Idea: approximate C and $\vec{\nabla}_{\vec{p}} C$ by summing over a randomly-chosen "mini-batch" $\mathcal{B}$ of $M_{\mathcal{B}}$ data points



$= 0$ for $k \leq N$

∂p_k

descent (GD):

$\eta \vec{\nabla}_{\vec{p}} C$

η is the learning rate

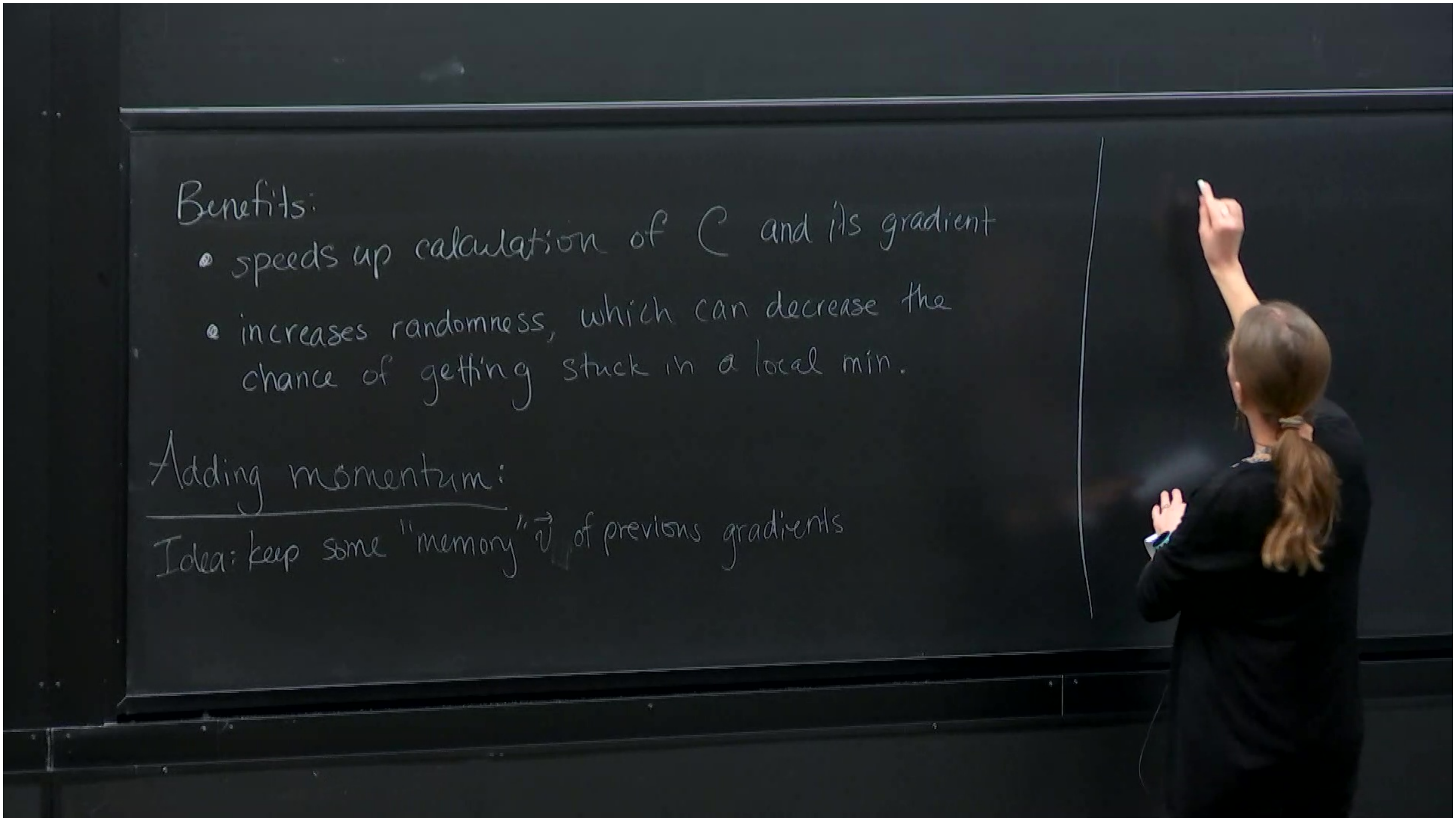
alternatives/improvements to GD

Stochastic GD (SGD):

Assume the cost func. can be written as

$$C = \frac{1}{|\mathcal{D}|} \sum_{\vec{x} \in \mathcal{D}} c(\vec{x})$$

Idea: approximate C and $\vec{\nabla}_{\vec{p}} C$ by summing over a randomly-chosen "mini-batch" $\mathcal{B}$ of $M_{\mathcal{B}}$ data points: $\sum_{\vec{x} \in \mathcal{D}} \longrightarrow \sum_{\vec{x} \in \mathcal{B}}$

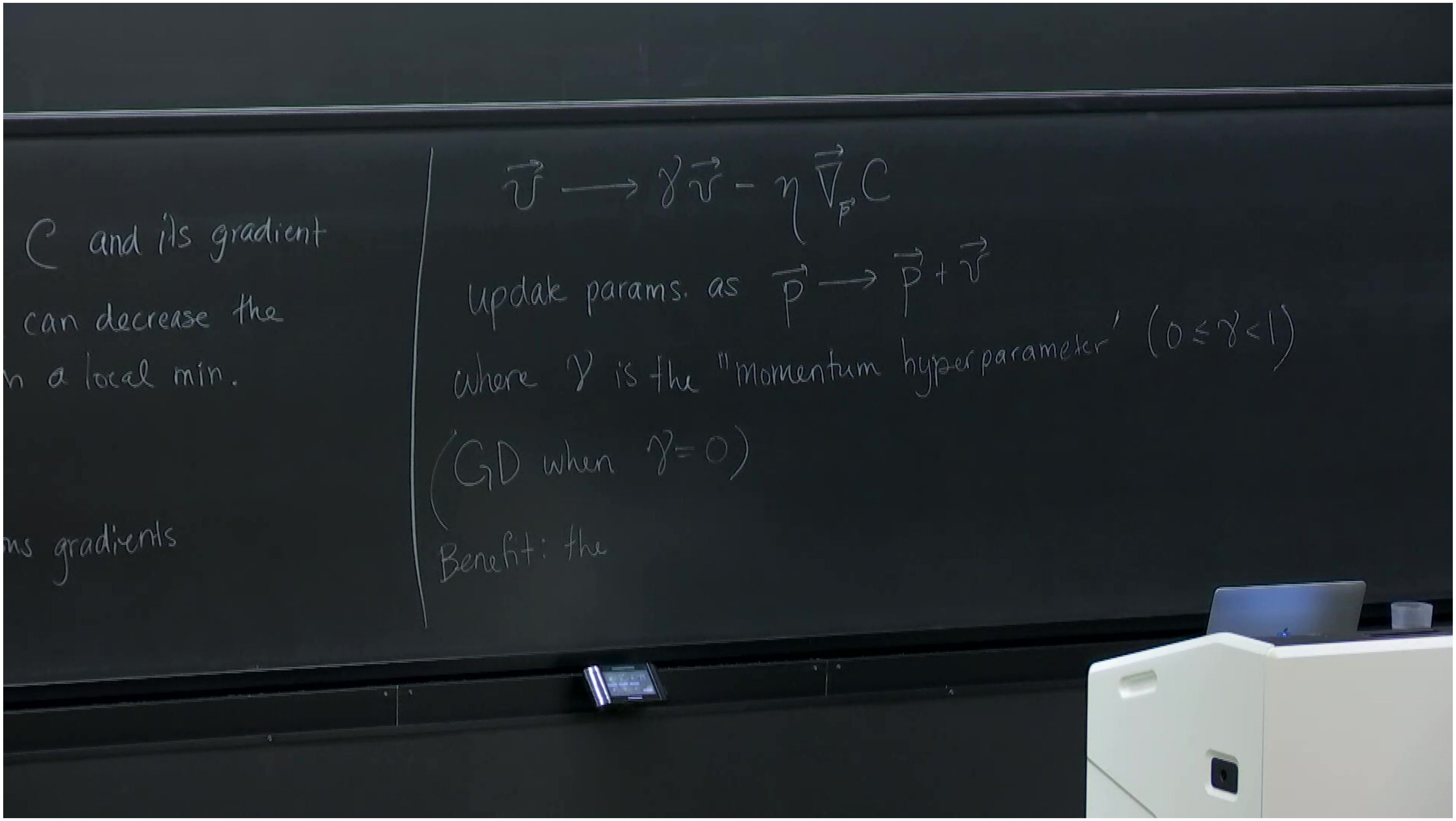


Benefits:

- speeds up calculation of C and its gradient
- increases randomness, which can decrease the chance of getting stuck in a local min.

Adding momentum:

Idea: keep some "memory" $\vec{v}$ of previous gradients



C and its gradient

can decrease the

n a local min.

ns gradients

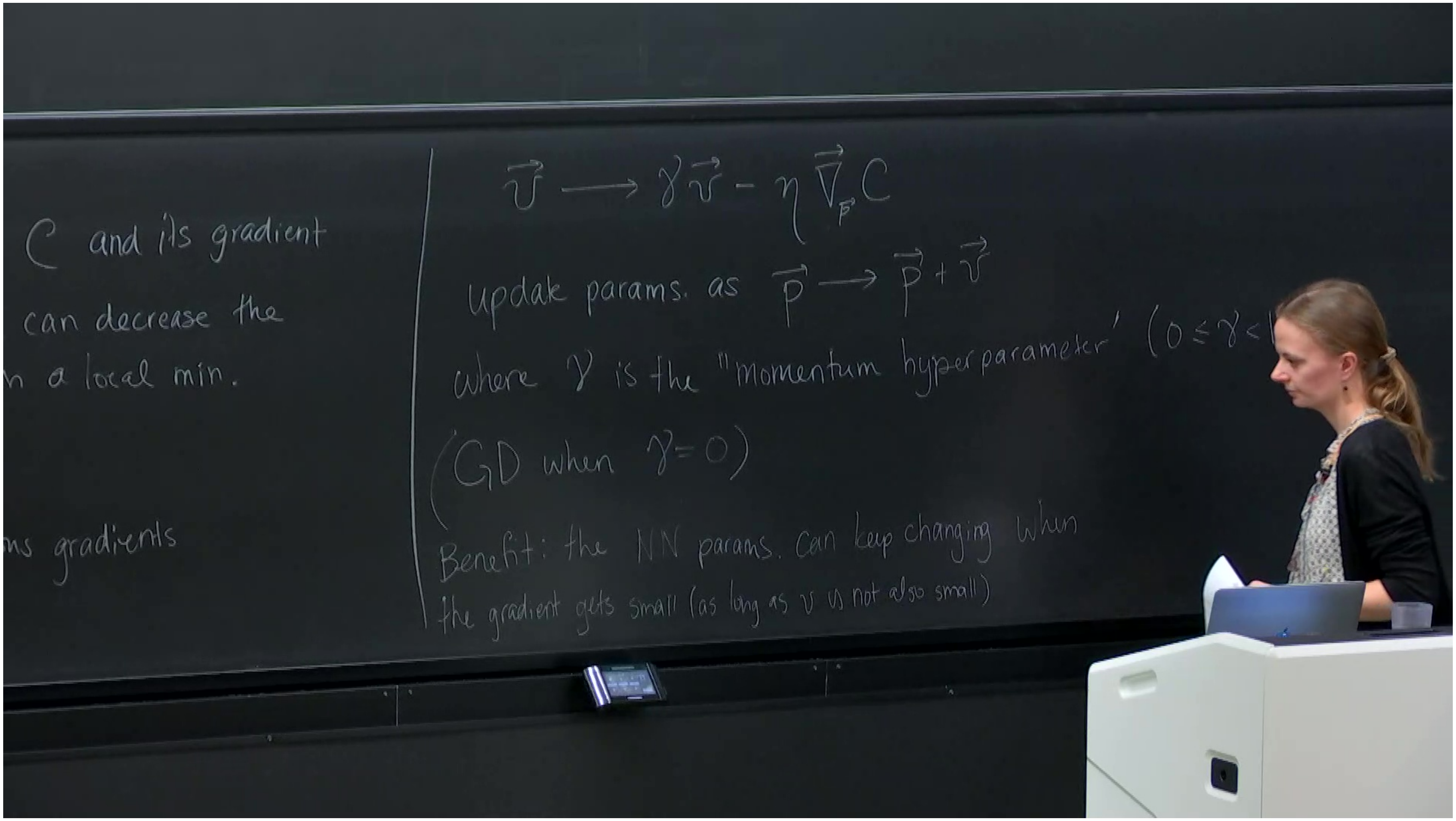
$$\vec{v} \longrightarrow \gamma \vec{v} - \eta \vec{\nabla}_{\vec{p}} C$$

update params. as $\vec{p} \longrightarrow \vec{p} + \vec{v}$

where γ is the "momentum hyperparameter" $(0 \le \gamma < 1)$

(GD when $\gamma = 0$)

Benefit: the



C and its gradient

can decrease the

n a local min.

ns gradients

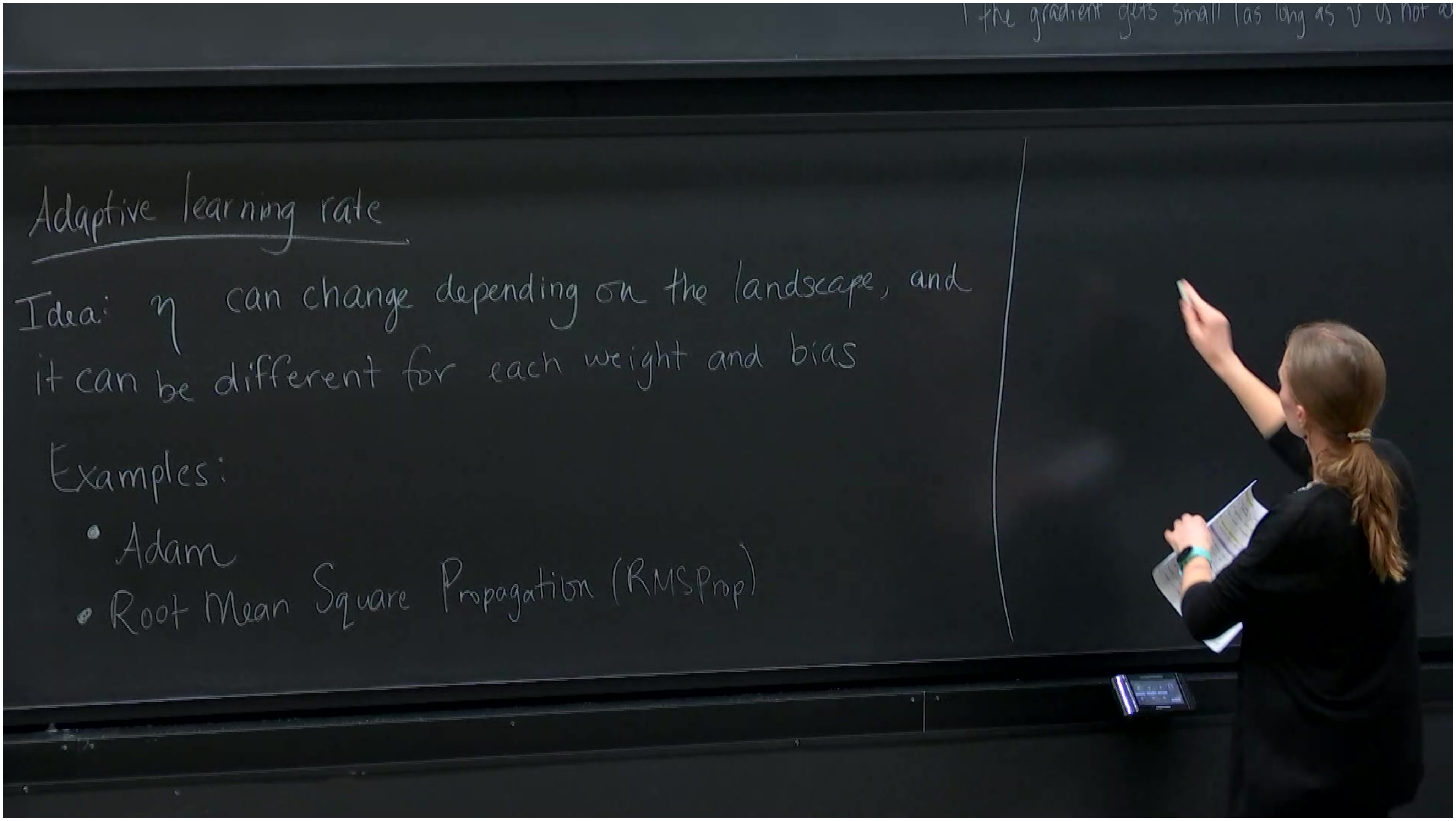
$$\vec{v} \longrightarrow \gamma \vec{v} - \eta \vec{\nabla}_{\vec{p}} C$$

update params. as $\vec{p} \longrightarrow \vec{p} + \vec{v}$

where γ is the "momentum hyperparameter" ($0 \leq \gamma <$

(GD when $\gamma = 0$)

Benefit: the NN params. can keep changing when the gradient gets small (as long as v is not also small)



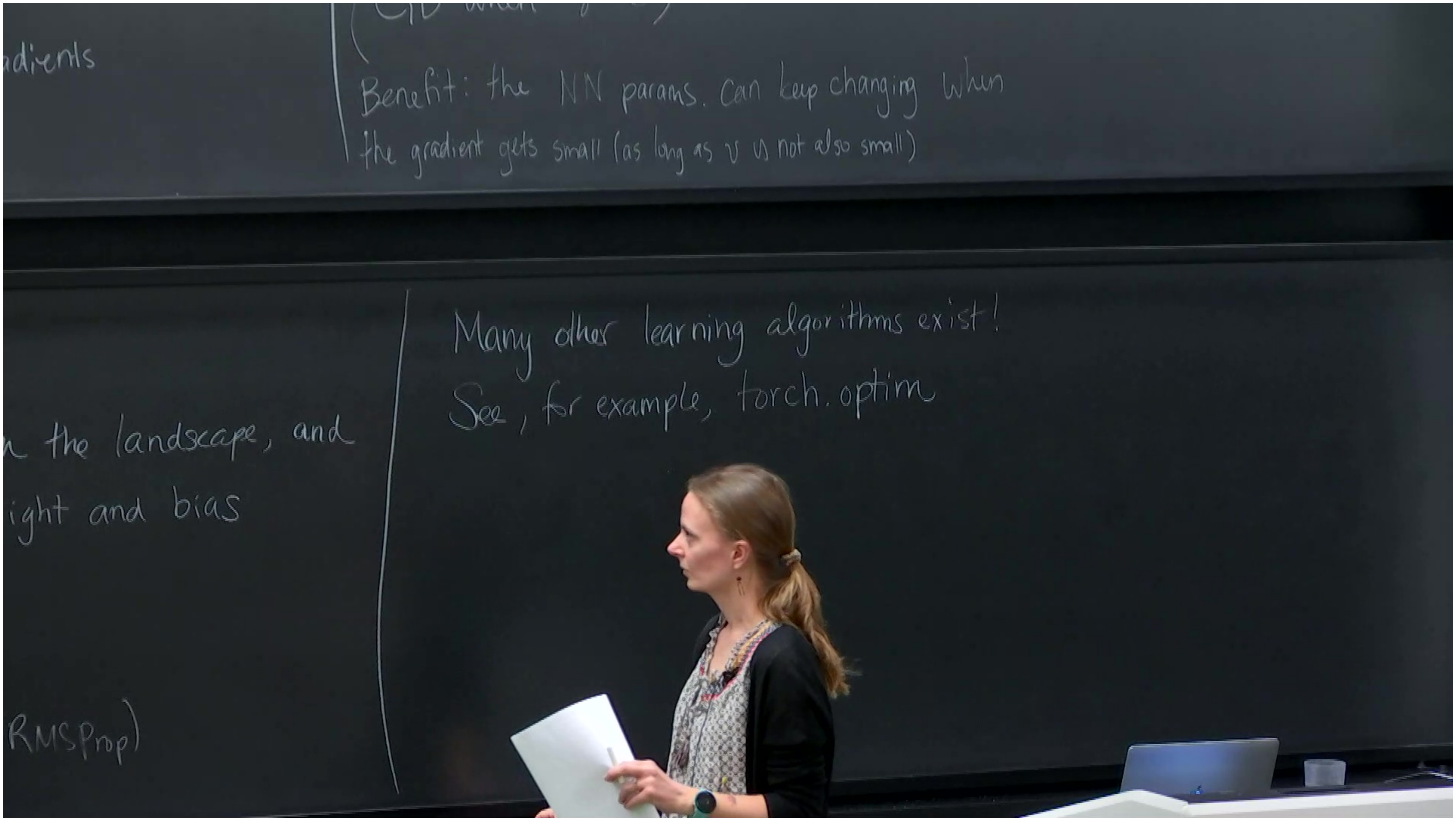
the gradient gets small (as long as v is not a

Adaptive learning rate

Idea: η can change depending on the landscape, and it can be different for each weight and bias

Examples:

- Adam
- Root Mean Square Propagation (RMSProp)



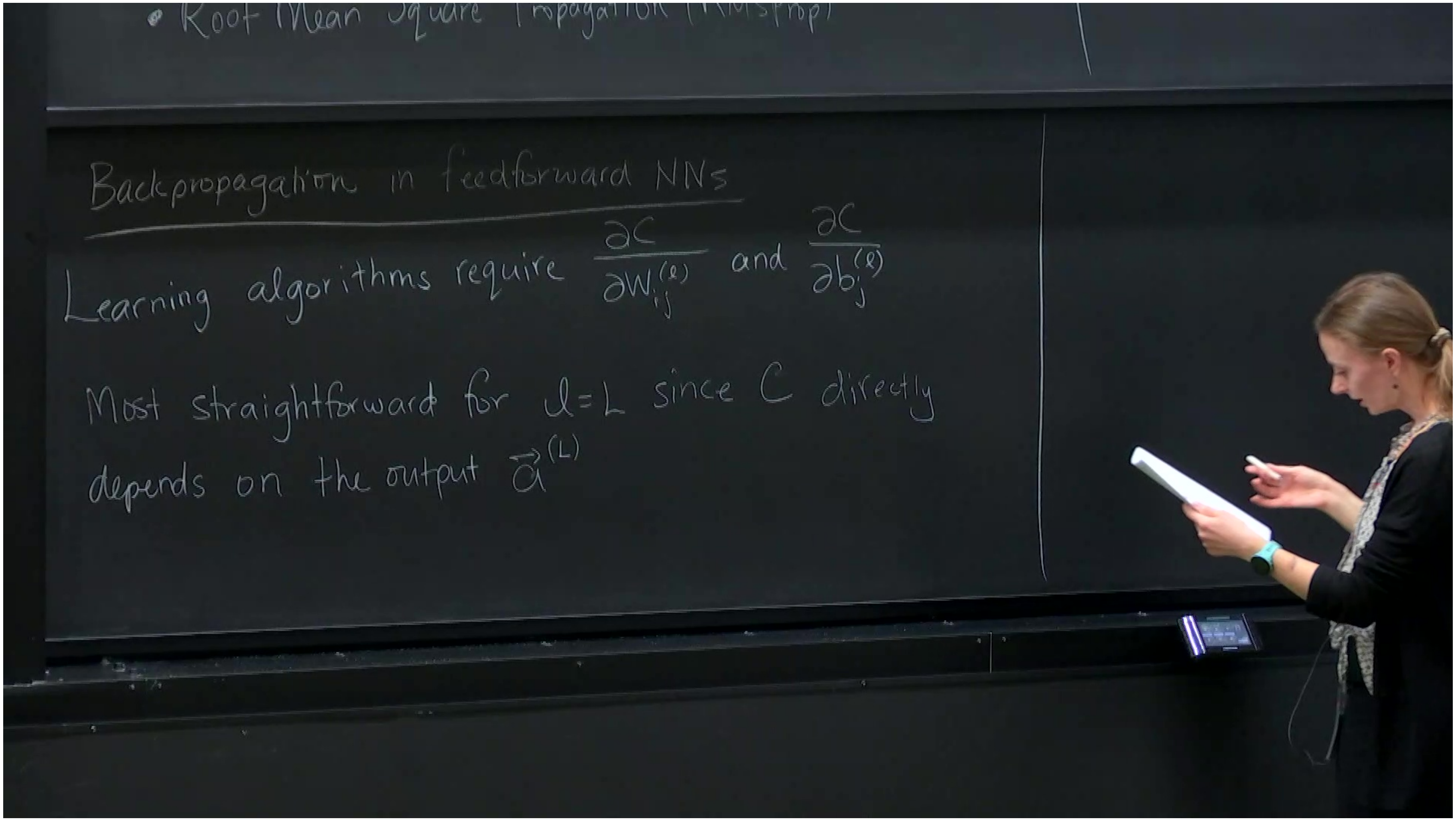
adients

Benefit: the NN params. can keep changing when the gradient gets small (as long as v is not also small)

Many other learning algorithms exist!
See, for example, torch.optim

n the landscape, and
ight and bias

RMSProp)

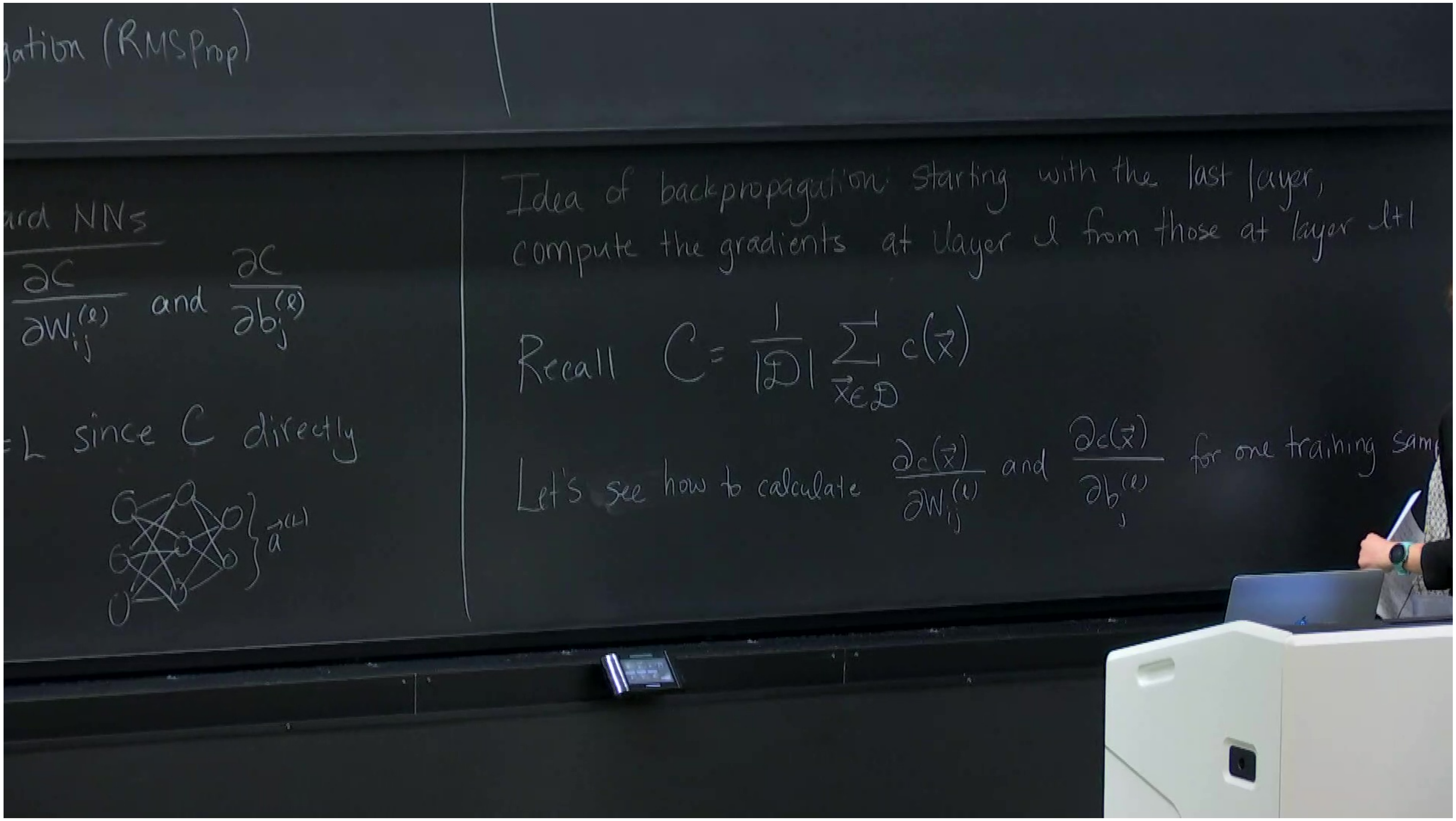


• Root Mean Square Propagation (RMSprop)

Backpropagation in feedforward NNs

Learning algorithms require $\frac{\partial C}{\partial w_{ij}^{(\ell)}}$ and $\frac{\partial C}{\partial b_j^{(\ell)}}$

Most straightforward for $\ell = L$ since C directly depends on the output $\vec{a}^{(L)}$



gation (RMSProp)

ard NNs

$\frac{\partial C}{\partial w_{ij}^{(\ell)}}$ and $\frac{\partial C}{\partial b_j^{(\ell)}}$

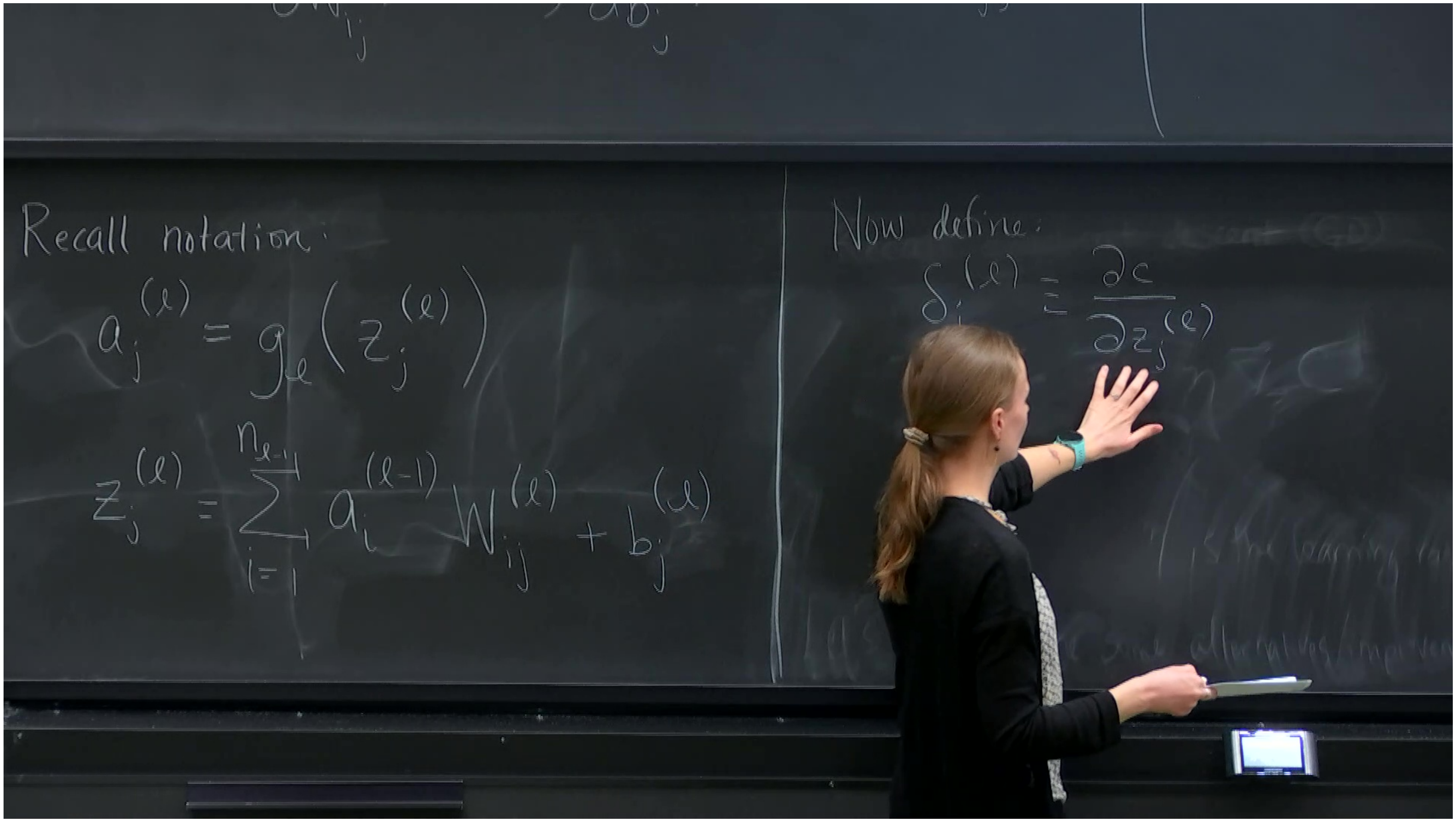
L since C directly

$\Big\}\ \vec{a}^{(L)}$

Idea of backpropagation: starting with the last layer, compute the gradients at layer ℓ from those at layer $\ell+1$

Recall $C = \frac{1}{|\mathcal{D}|} \sum_{\vec{x} \in \mathcal{D}} c(\vec{x})$

Let's see how to calculate $\frac{\partial c(\vec{x})}{\partial w_{ij}^{(\ell)}}$ and $\frac{\partial c(\vec{x})}{\partial b_j^{(\ell)}}$ for one training sam



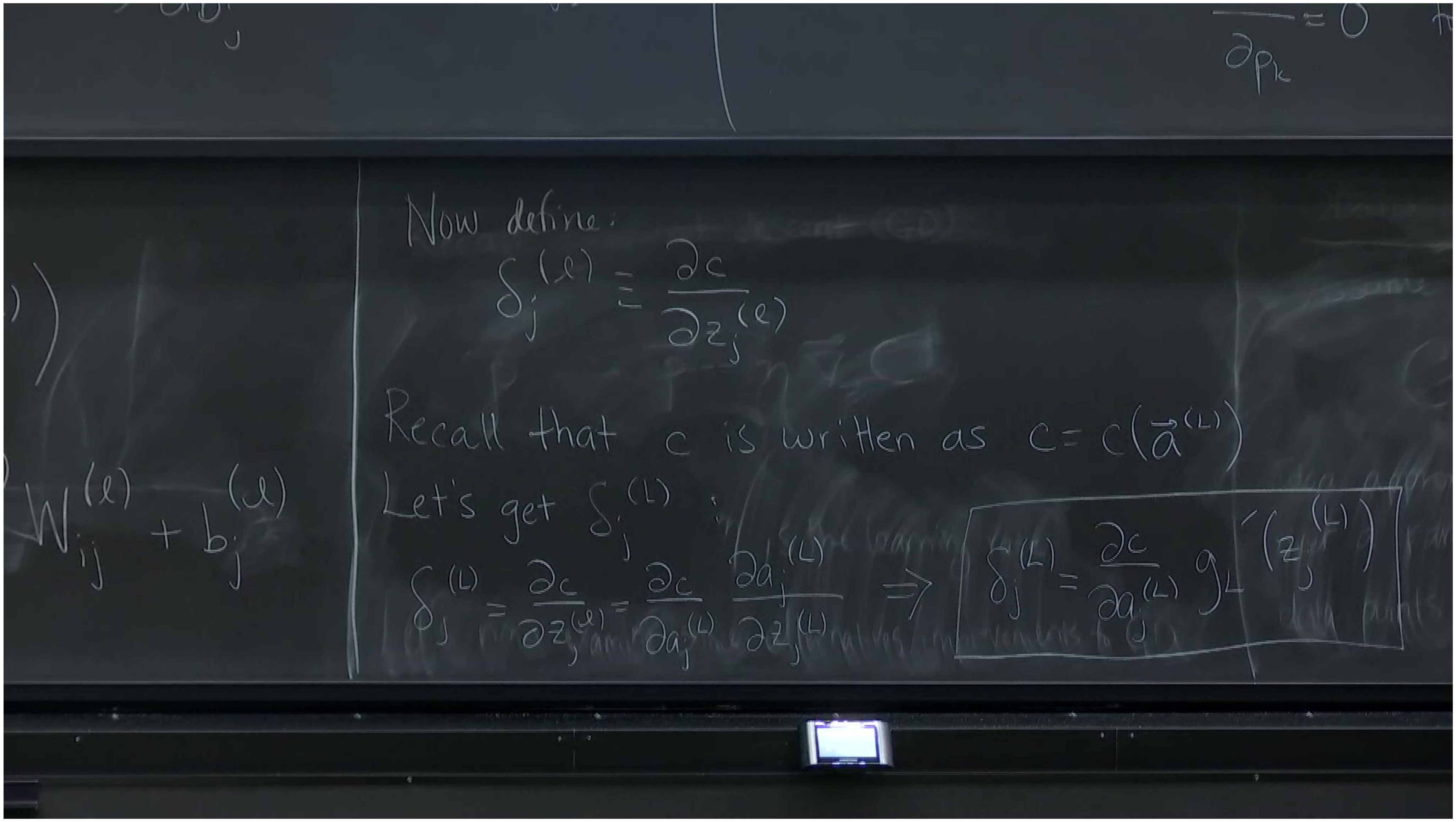
Recall notation:

$$a_j^{(\ell)} = g_\ell\left(z_j^{(\ell)}\right)$$

$$z_j^{(\ell)} = \sum_{i=1}^{n_{\ell-1}} a_i^{(\ell-1)} W_{ij}^{(\ell)} + b_j^{(\ell)}$$

Now define:

$$\delta_j^{(\ell)} \equiv \frac{\partial C}{\partial z_j^{(\ell)}}$$



$W_{ij}^{(\ell)} + b_j^{(\ell)}$

Now define:

$$\delta_j^{(\ell)} \equiv \frac{\partial c}{\partial z_j^{(\ell)}}$$

Recall that c is written as $c = c(\vec{a}^{(L)})$

Let's get $\delta_j^{(L)}$:

$$\delta_j^{(L)} = \frac{\partial c}{\partial z_j^{(\ell)}} = \frac{\partial c}{\partial a_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \Rightarrow \boxed{\delta_j^{(L)} = \frac{\partial c}{\partial a_j^{(L)}} g_L'(z_j^{(L)})}$$

$\frac{\quad}{\partial p_k} = 0$ for $1 \leq k$

$+ b_j^{(l)}$

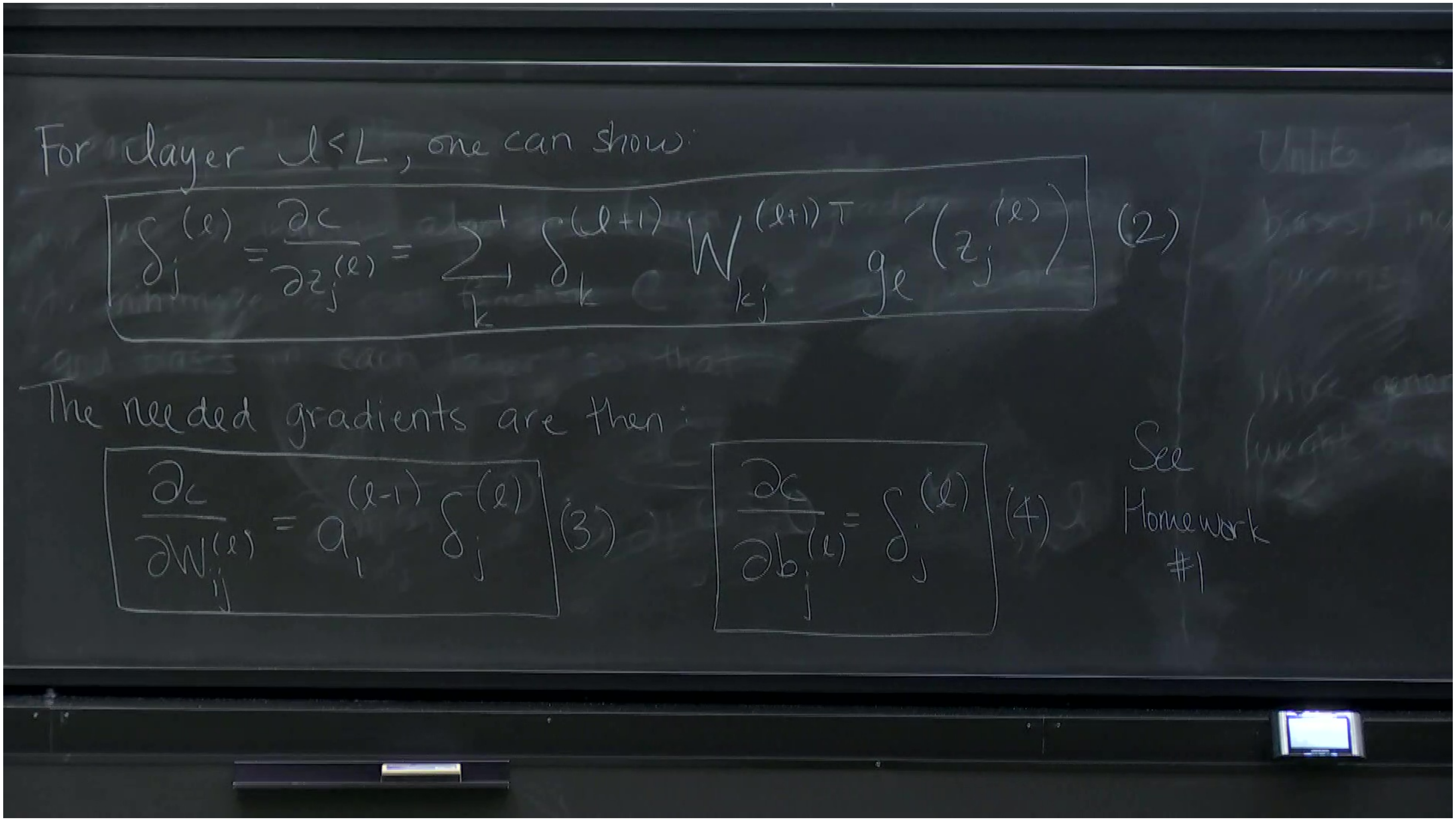
Now define:

$$\delta_j^{(l)} \equiv \frac{\partial c}{\partial z_j^{(l)}}$$

Recall that c is written as $c = c(\vec{a}^{(L)})$

Let's get $\delta_j^{(L)}$:

$$\delta_j^{(L)} = \frac{\partial c}{\partial z_j^{(L)}} = \frac{\partial c}{\partial a_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \Rightarrow \boxed{\delta_j^{(L)} = \frac{\partial c}{\partial a_j^{(L)}} g_L'(z_j^{(L)})} \quad (1)$$



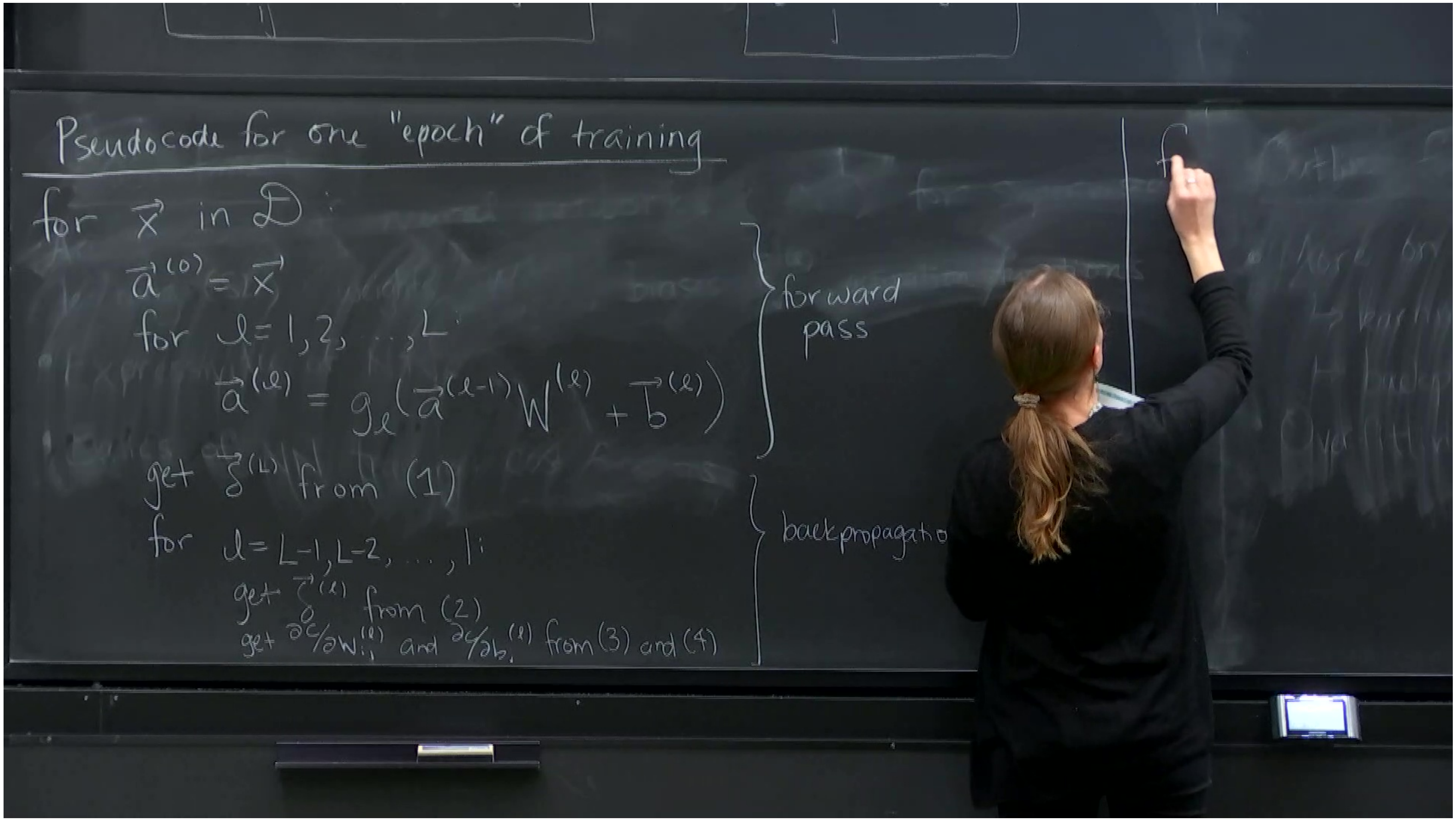
For layer $\ell < L$, one can show:

$$\boxed{\delta_j^{(\ell)} = \frac{\partial c}{\partial z_j^{(\ell)}} = \sum_k \delta_k^{(\ell+1)} W_{kj}^{(\ell+1)T} g_\ell'\left(z_j^{(\ell)}\right)} \quad (2)$$

The needed gradients are then:

$$\boxed{\frac{\partial c}{\partial W_{ij}^{(\ell)}} = a_i^{(\ell-1)} \delta_j^{(\ell)}} \quad (3) \qquad \boxed{\frac{\partial c}{\partial b_j^{(\ell)}} = \delta_j^{(\ell)}} \quad (4)$$

See Homework #1



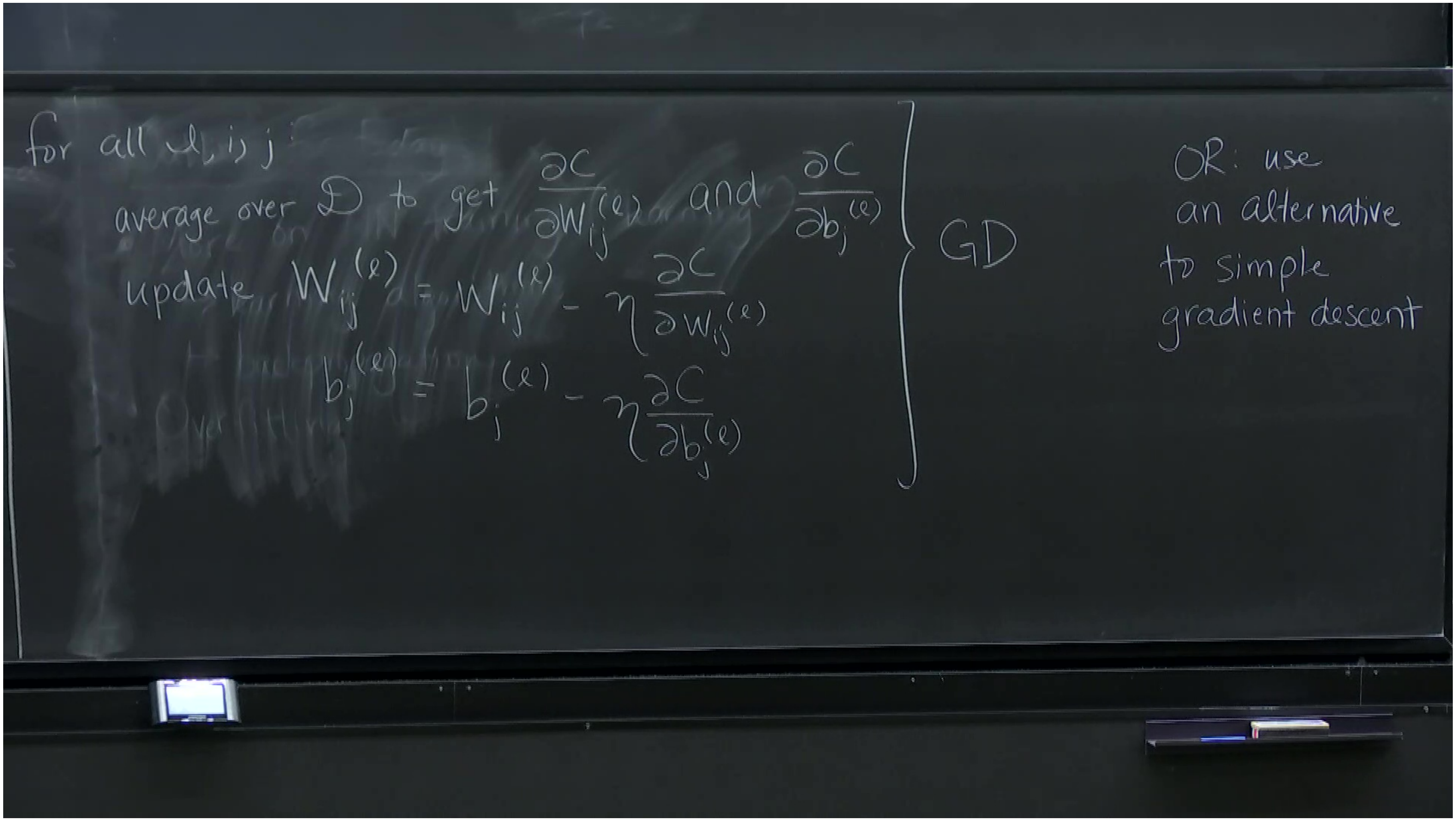
Pseudocode for one "epoch" of training

```
for $\vec{x}$ in $\mathcal{D}$:
    $\vec{a}^{(0)} = \vec{x}$
    for $l = 1, 2, \ldots, L$:
        $\vec{a}^{(l)} = g_l(\vec{a}^{(l-1)} W^{(l)} + \vec{b}^{(l)})$
    get $\vec{\delta}^{(L)}$ from (1)
    for $l = L-1, L-2, \ldots, 1$:
        get $\vec{\delta}^{(l)}$ from (2)
        get $\partial C / \partial W_{ij}^{(l)}$ and $\partial C / \partial b_i^{(l)}$ from (3) and (4)
```

forward pass

backpropagatio

f



for all ℓ, i, j:

average over $\mathcal{D}$ to get $\frac{\partial C}{\partial w_{ij}^{(\ell)}}$ and $\frac{\partial C}{\partial b_j^{(\ell)}}$

update $W_{ij}^{(\ell)} = W_{ij}^{(\ell)} - \eta \frac{\partial C}{\partial w_{ij}^{(\ell)}}$

$b_j^{(\ell)} = b_j^{(\ell)} - \eta \frac{\partial C}{\partial b_j^{(\ell)}}$

$\Big\}$ GD

OR: use an alternative to simple gradient descent

Overfitting

Training data:

```
N_train = 10
x = np.linspace(0.05,0.95,N_train)
s = np.random.randn(N_train)
y = 2*x+s
```

Models for fitting:

Polynomials of degree 1, 3, and 10

Question: Which model will give the best fit (lowest error) to the training data?

Overfitting

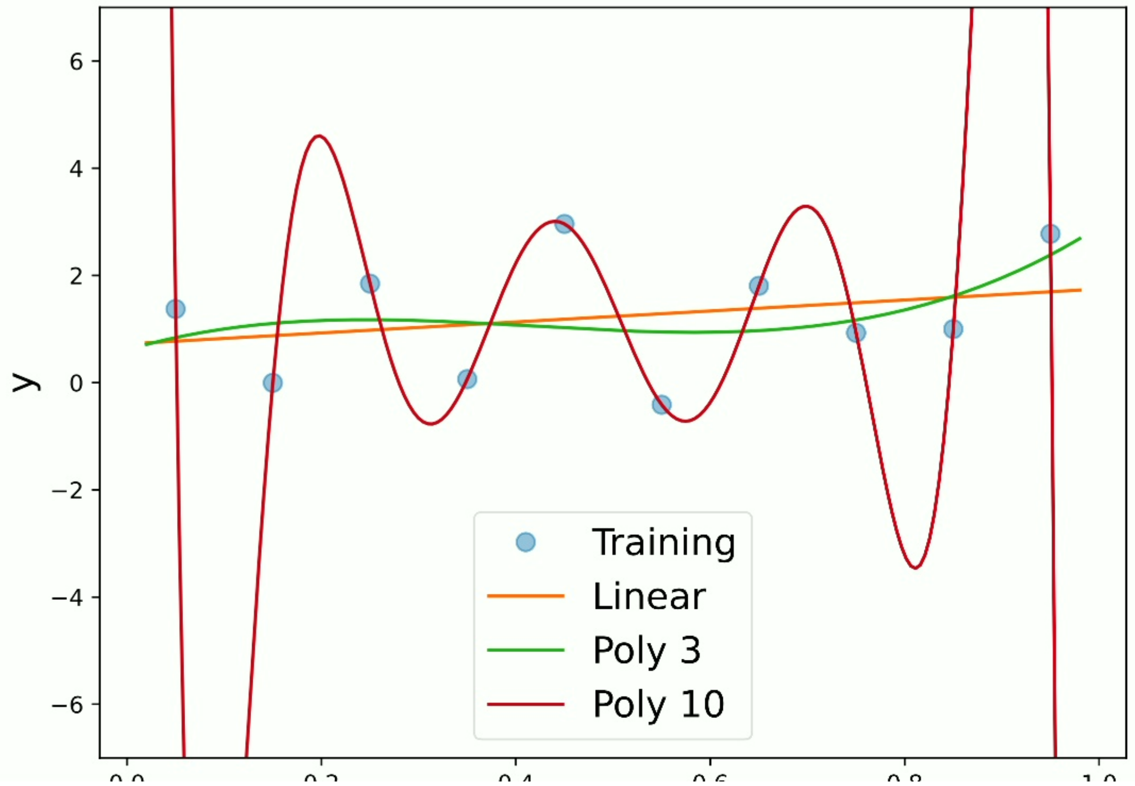
Training data:

```
N_train = 10
x = np.linspace(0.05,0.95,N_train)
s = np.random.randn(N_train)
y = 2*x+s
```

Models for fitting:

Polynomials of degree 1, 3, and 10

Question: Which model will give the best fit (lowest error) to the training data?



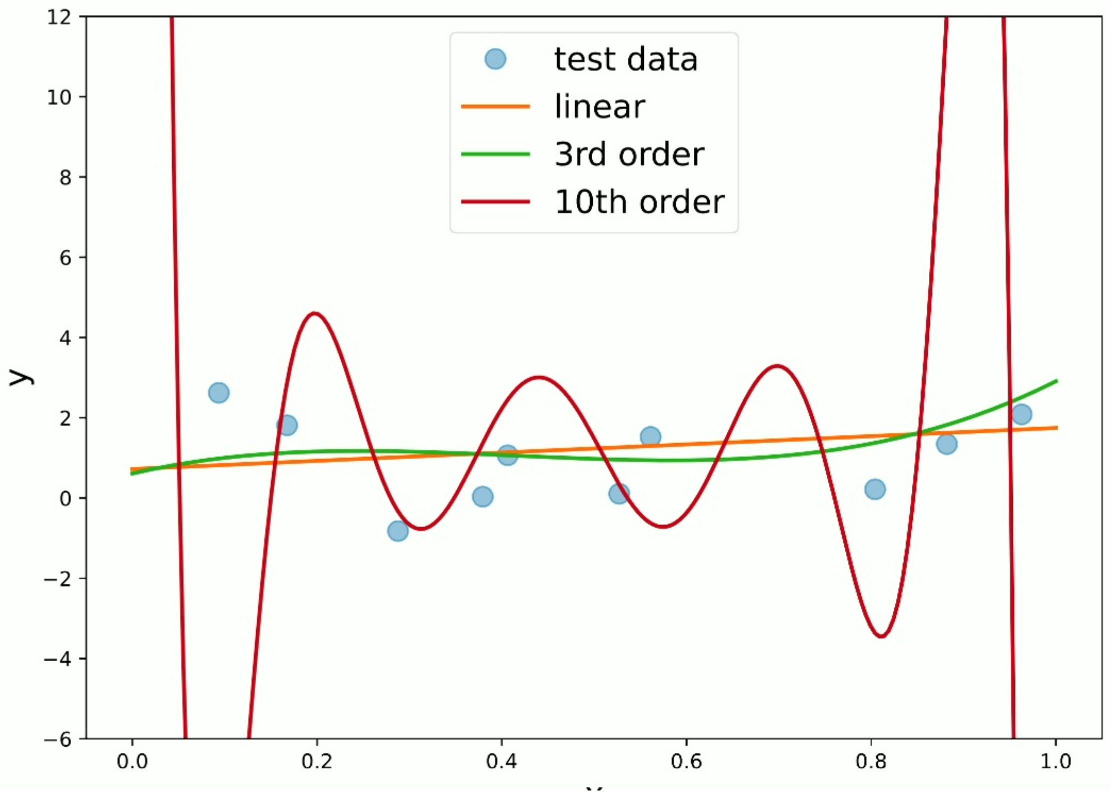
Overfitting

Testing data (from the same distribution as the training data):

```
N_test = 10
x_test = np.random.random(N_test)
s_test = np.random.randn(N_test)
y_test = 2*x_test+s_test
```

Questions:

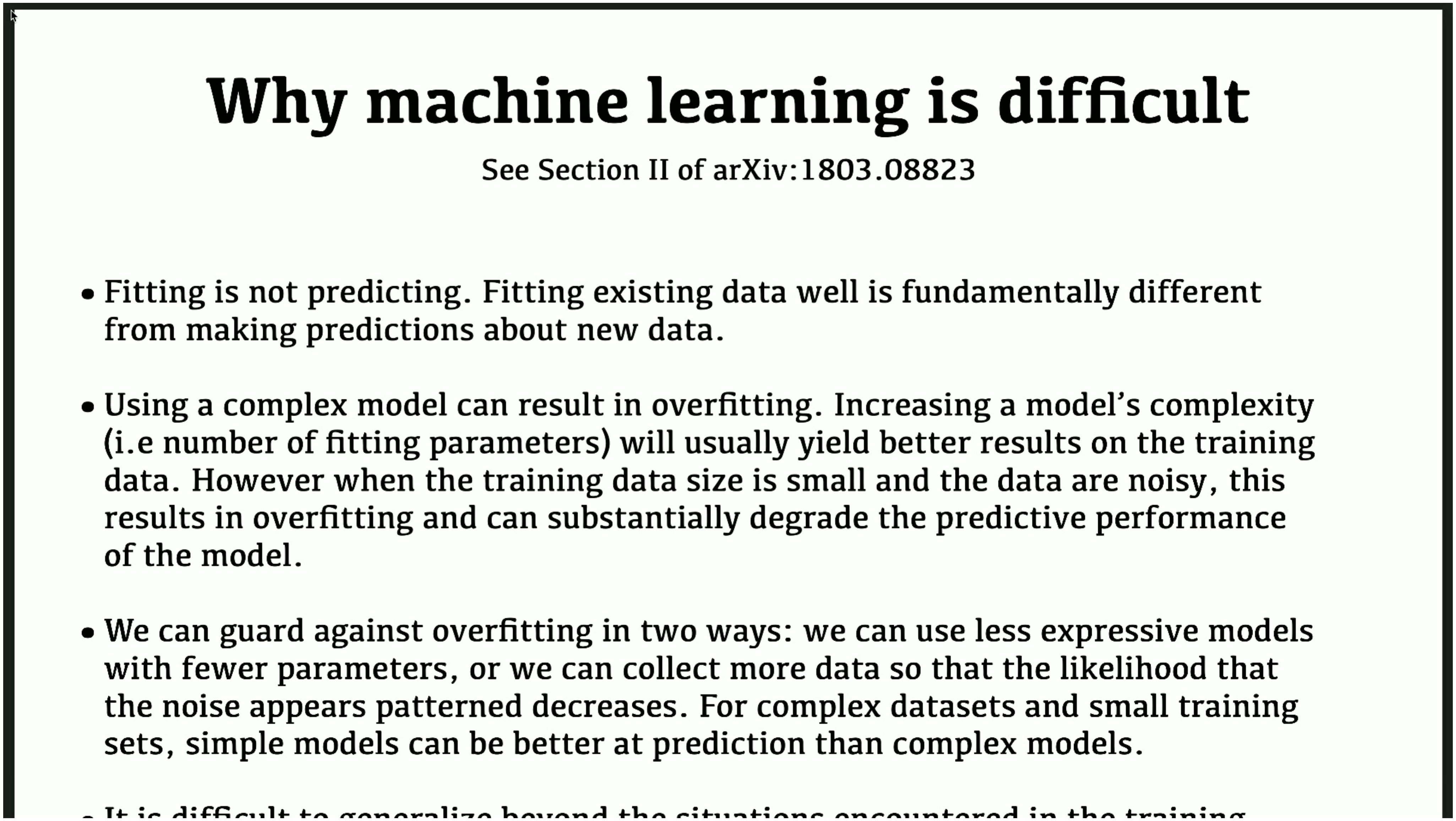
- Which model from the previous slide will make the best predictions on the testing data?
- What would happen if we increased the number of data points in the training dataset?



Why machine learning is difficult

See Section II of arXiv:1803.08823

- Fitting is not predicting. Fitting existing data well is fundamentally different from making predictions about new data.
- Using a complex model can result in overfitting. Increasing a model's complexity (i.e number of fitting parameters) will usually yield better results on the training data. However when the training data size is small and the data are noisy, this results in overfitting and can substantially degrade the predictive performance of the model.
- We can guard against overfitting in two ways: we can use less expressive models with fewer parameters, or we can collect more data so that the likelihood that the noise appears patterned decreases. For complex datasets and small training sets, simple models can be better at prediction than complex models.



Why machine learning is difficult

See Section II of arXiv:1803.08823

- Fitting is not predicting. Fitting existing data well is fundamentally different from making predictions about new data.
- Using a complex model can result in overfitting. Increasing a model's complexity (i.e number of fitting parameters) will usually yield better results on the training data. However when the training data size is small and the data are noisy, this results in overfitting and can substantially degrade the predictive performance of the model.
- We can guard against overfitting in two ways: we can use less expressive models with fewer parameters, or we can collect more data so that the likelihood that the noise appears patterned decreases. For complex datasets and small training sets, simple models can be better at prediction than complex models.
- It is difficult to generalize beyond the situations encountered in the training