

CS/ENGR M148 L16: NN Regularization

Sandra Batista

Holiday break this week!

PS 4 posted and due 12/3/24

Final project specifications posted

Extra credit Final Exam Review Question Code Bank:

<https://forms.gle/XdC97wxwWd8QuTR9A>

Questions due by 11:59 pm PT on 11/26/24.

We'll share questions with solutions during week 10 for final exam review.

Final project Specs

You can share your project work publicly as part of portfolios.

1. Report [50 points]: Main document and Appendix
2. **Code** [50 points]: Main Jupyter Notebook for entire project pipeline that work (including data, loading, EDA, and analyses) and then auxiliary notebooks for work in Appendix

Report

1. Main document: Describe data set, problem addressed, key methods, results and conclusions, and how to use your project code.
2. Appendix: Discuss all methods from check-in and why they did or did not work on your project.

Join our slido for the week...

<https://app.sli.do/event/nCV57u4mC7eUMit9euSBr2>



Today's Learning Objectives

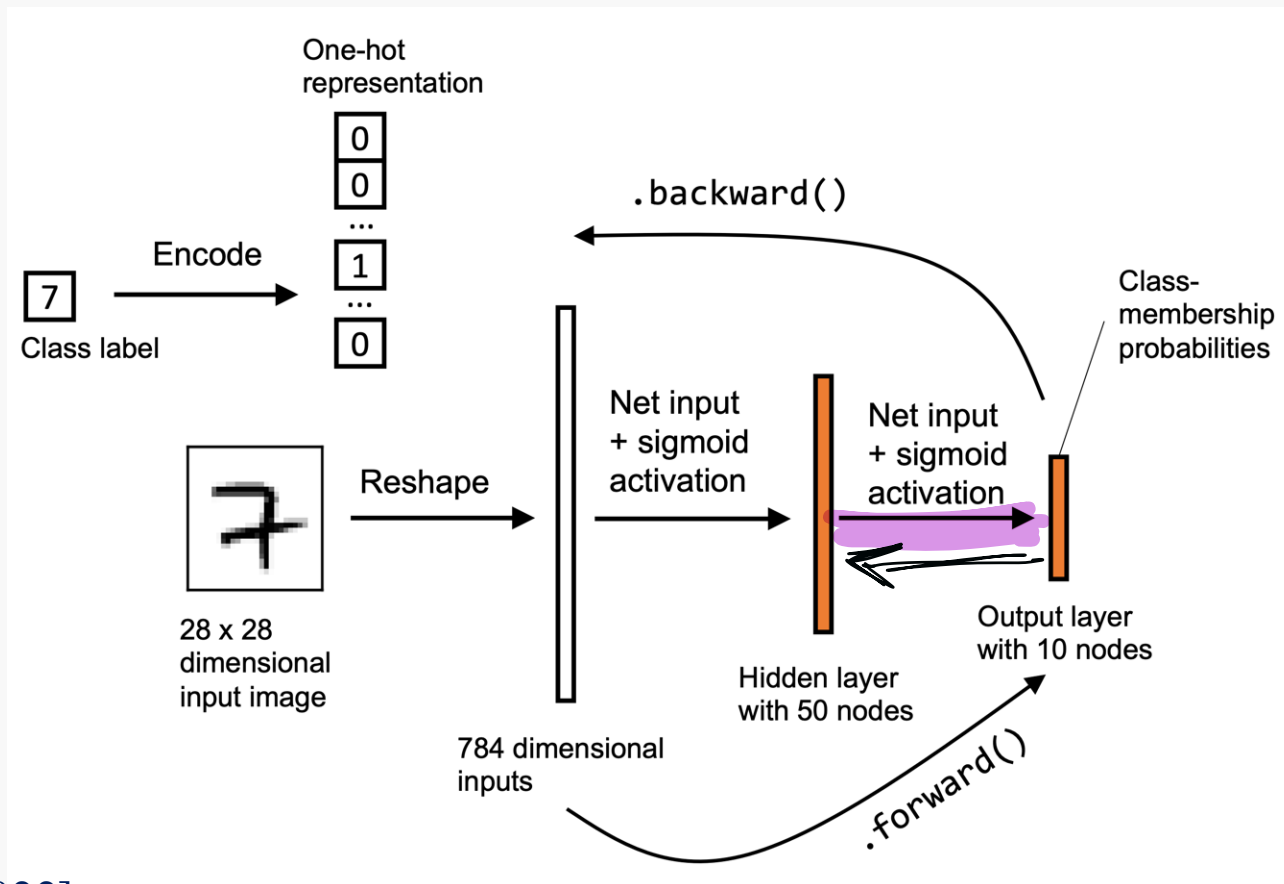
Students will be able to:

- Review: Backprop on NN with MNIST Code
- Review: Understand the role of gradient descent in training NN with MNIST Code
- Regularization for NN

MNIST NN

Today we'll work on forward algorithm..

Next lecture backpropagation, gradient descent, training, and evaluating network performance



Loss Functions

As with other models, we have choice of loss functions, such as

- 1. MSE*
- 2. Negative Log Likelihood (Binary Cross Entropy) or its generalization for multi-class classification*

We'll look at MSE for MNIST

Your turn: MNIST NN

Please get the Jupyter notebook for MNIST:

https://colab.research.google.com/drive/10BkOu_bVBs3af2NXFrD48iK8s3MJIScj?usp=sharing

Save a copy to your Google Drive and keep notes there...

Backprop Algorithm

1. *Compute the gradient of loss function with respect to output layer*
2. *For each layer*
 - i. *Convert gradient on layer's output into a gradient with respect to the net input before activation*
 - ii. *Compute gradients on weights and biases by using gradient of the net input with respect to weights*
 - iii. *Propagate the gradients backwards in NN (i.e. save to reuse for gradients in lower layers)*

***Gradients give us how much output layer or weight should change
To reduce loss.***

60

MNIST Backprop

```
def backward(self, x, a_h, a_out, y):
```

```
#####  
### Output layer weights  
#####
```

```
# onehot encoding  
y_onehot = int_to_onehot(y, self.num_classes)
```

```
# Part 1: dLoss/dOutWeights  
## = dLoss/dOutAct * dOutAct/dOutNet * dOutNet/dOutWeight  
## where DeltaOut = dLoss/dOutAct * dOutAct/dOutNet  
## for convenient re-use
```

```
* # input/output dim: [n_examples, n_classes]  
d_loss__d_a_out = 2.*(a_out - y_onehot) / y.shape[0]
```

```
# input/output dim: [n_examples, n_classes]  
d_a_out__d_z_out = a_out * (1. - a_out) # sigmoid derivative
```

```
# output dim: [n_examples, n_classes]  
delta_out = d_loss__d_a_out * d_a_out__d_z_out # "delta (rule) placeholder"
```

```
# gradient for output weights
```

```
* # [n_examples, n_hidden]  
d_z_out__dw_out = a_h
```

```
# input dim: [n_classes, n_examples] dot [n_examples, n_hidden]  
# output dim: [n_classes, n_hidden]  
d_loss__dw_out = np.dot(delta_out.T, d_z_out__dw_out)  
d_loss__db_out = np.sum(delta_out, axis=0)
```

activation function
(sigmoid)

$$a^{out} = \sigma(z^{out})$$

→ derivative of mse with respect to a^{out}

$$\sigma(z^{out}) (1 - \sigma(z^{out}))$$

$$\delta = \frac{\partial L}{\partial a^{out}} \frac{\partial a^{out}}{\partial z^{out}}$$

$$z^{out} = W^{out} a^h + b^{out}$$

$$\frac{\partial L}{\partial W^{out}} = \delta \cdot \frac{\partial z^{out}}{\partial W^{out}}$$

[Raschka et al 2022]

↑
Notice that this

works because $\frac{\partial z^{(out)}}{\partial f^{(out)}} = 1$ and calculates $\frac{\partial L}{\partial b^{(out)}} = \delta \cdot \frac{\partial z^{(out)}}{\partial b^{(out)}}$

MNIST Backprop

```
#####
# Part 2: dLoss/dHiddenWeights
## = DeltaOut * dOutNet/dHiddenAct * dHiddenAct/dHiddenNet * dHiddenNet/dWeight
```

```
# [n_classes, n_hidden]
```

```
d_z_out__a_h = self.weight_out
```

(recall that z^{out} is linear combination of a^h)

```
# output dim: [n_examples, n_hidden]
```

```
d_loss__a_h = np.dot(delta_out, d_z_out__a_h)
```

```
# [n_examples, n_hidden]
```

```
d_a_h__d_z_h = a_h * (1. - a_h) # sigmoid derivative
```

```
# [n_examples, n_features]
```

```
d_z_h__d_w_h = x
```

(recall z^h is linear combination of input x)

```
# output dim: [n_hidden, n_features]
```

```
d_loss__d_w_h = np.dot((d_loss__a_h * d_a_h__d_z_h).T, d_z_h__d_w_h)
```

```
d_loss__d_b_h = np.sum((d_loss__a_h * d_a_h__d_z_h), axis=0)
```

```
return (d_loss__dw_out, d_loss__db_out,
        d_loss__d_w_h, d_loss__d_b_h)
```

Today's Learning Objectives

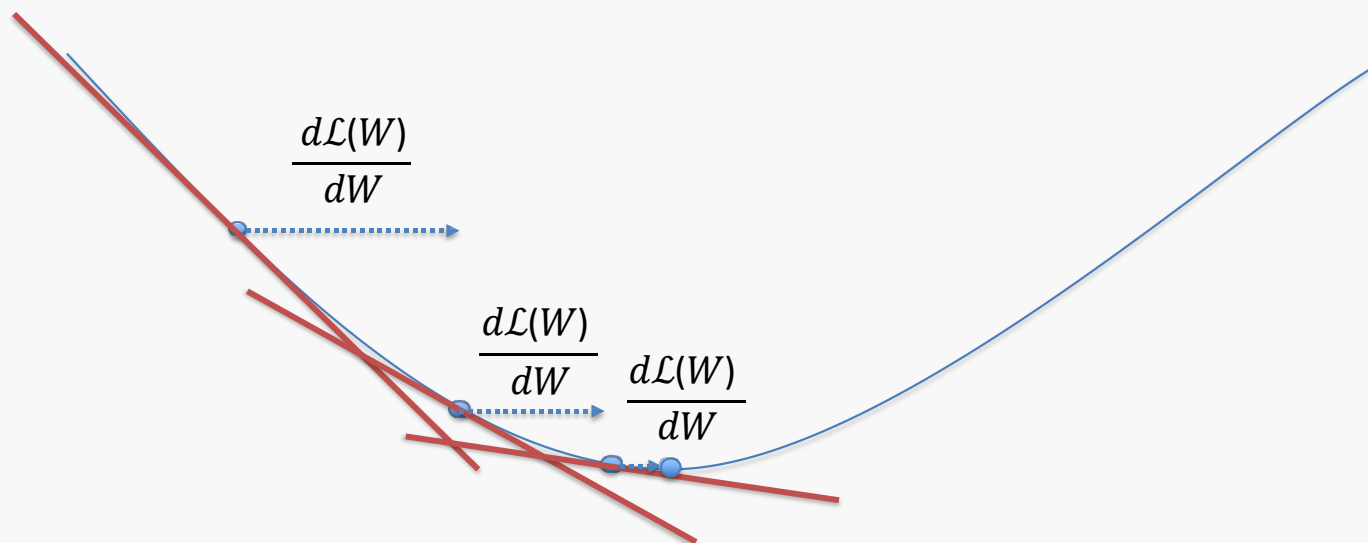
Students will be able to:

- ✓ Review: Backprop on NN with MNIST Code
 - Review: Understand the role of gradient descent in training NN with MNIST Code
 - Regularization for NN

Training to minimize the loss function

Gradient Descent

If the step is proportional to the slope then you avoid overshooting the minimum. How?



Gradient Descent

1. *Initialize small weights*
2. *For each sample*
 - i. *Apply the forward algorithm*
 - ii. *Apply backprop*
 - iii. *Use the gradients of the weights and biases to update the weights and biases*

*Repeat step 2 for a number of **epochs**, complete pass through training data*

Updating the weights

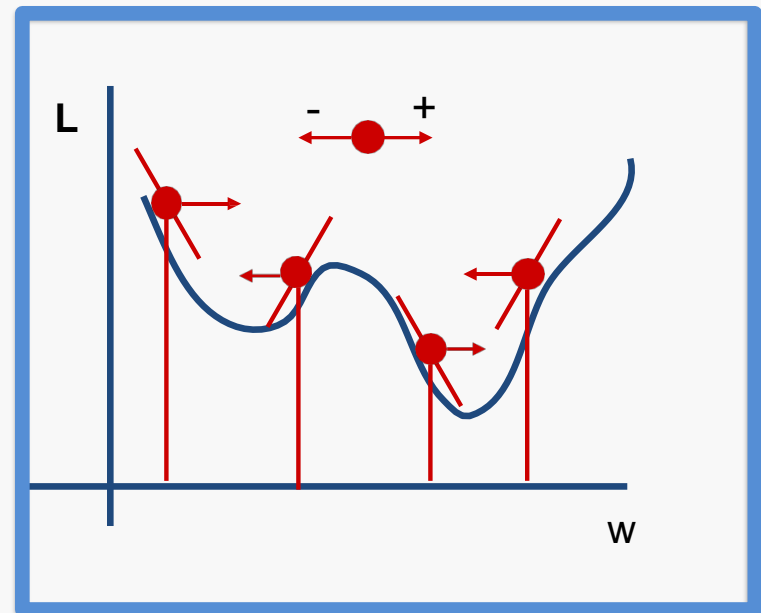
For each weight

$$\Delta w = \frac{\partial L}{\partial w}$$

$$w^{new} = w^{old} - \eta \Delta w$$

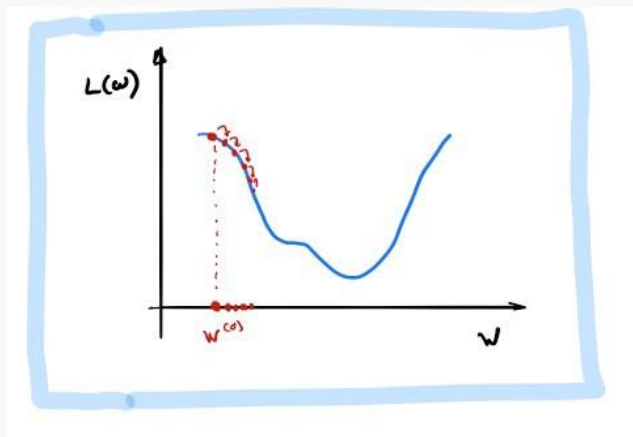
Gradient Descent

- Algorithm for optimization of first order to finding a minimum of a function.
- It is an iterative method.
- L is decreasing much faster in the direction of the negative derivative.
- The learning rate is controlled by the magnitude of η .
- Often find local minima not global

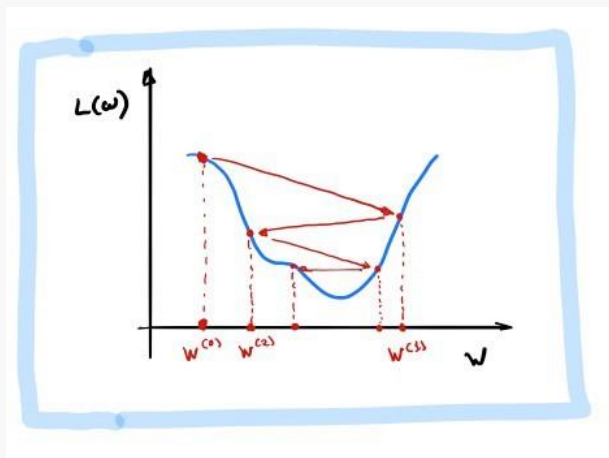


Learning Rate

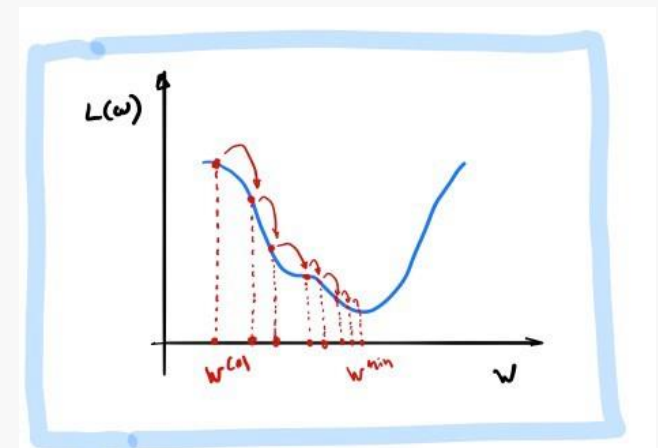
Our choice of the learning rate has a significant impact on the performance of gradient descent.



When η is too small, the algorithm makes very little progress.



When η is too large, the algorithm may overshoot the minimum and has crazy oscillations.



When η is appropriate, the algorithm will find the minimum. The algorithm **converges**!

Batch and Stochastic Gradient Descent

Instead of using all the training examples for every step,

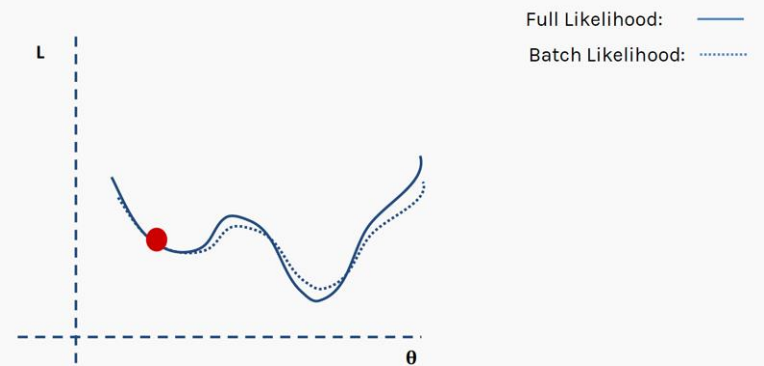
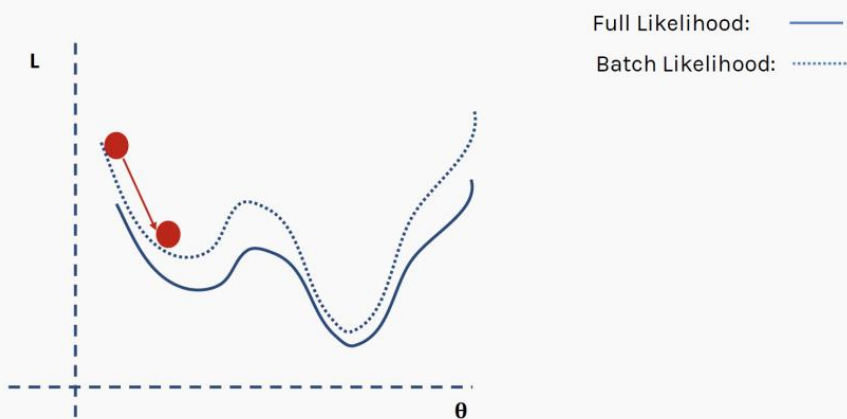
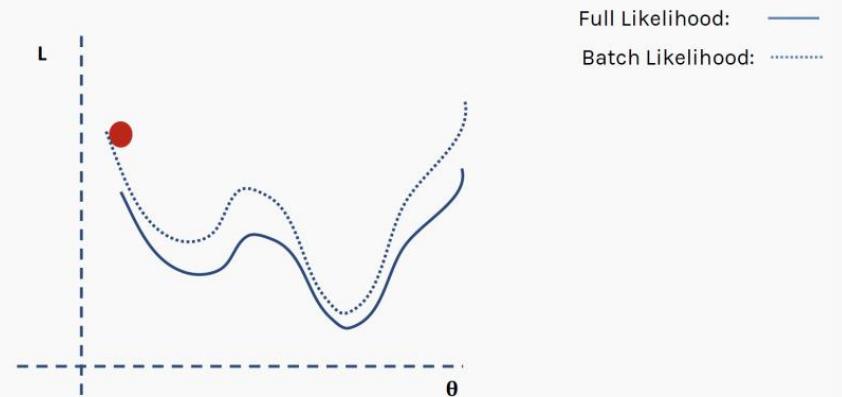
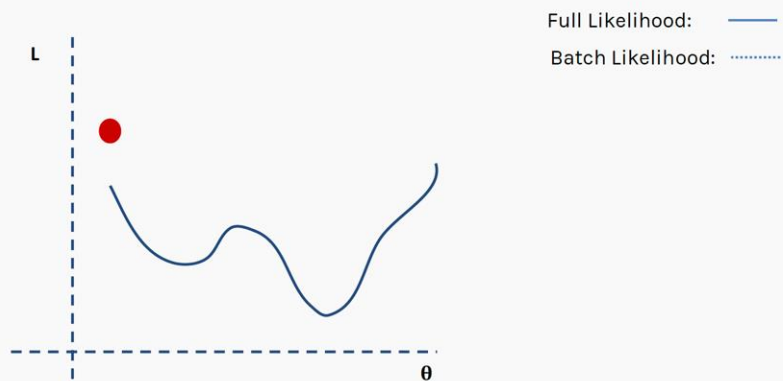
For **batch gradient descent**,

In each epoch use a subset or **batch** of samples. (These can be randomly selected by shuffling data). Use average of the gradients from the batches to update the weights

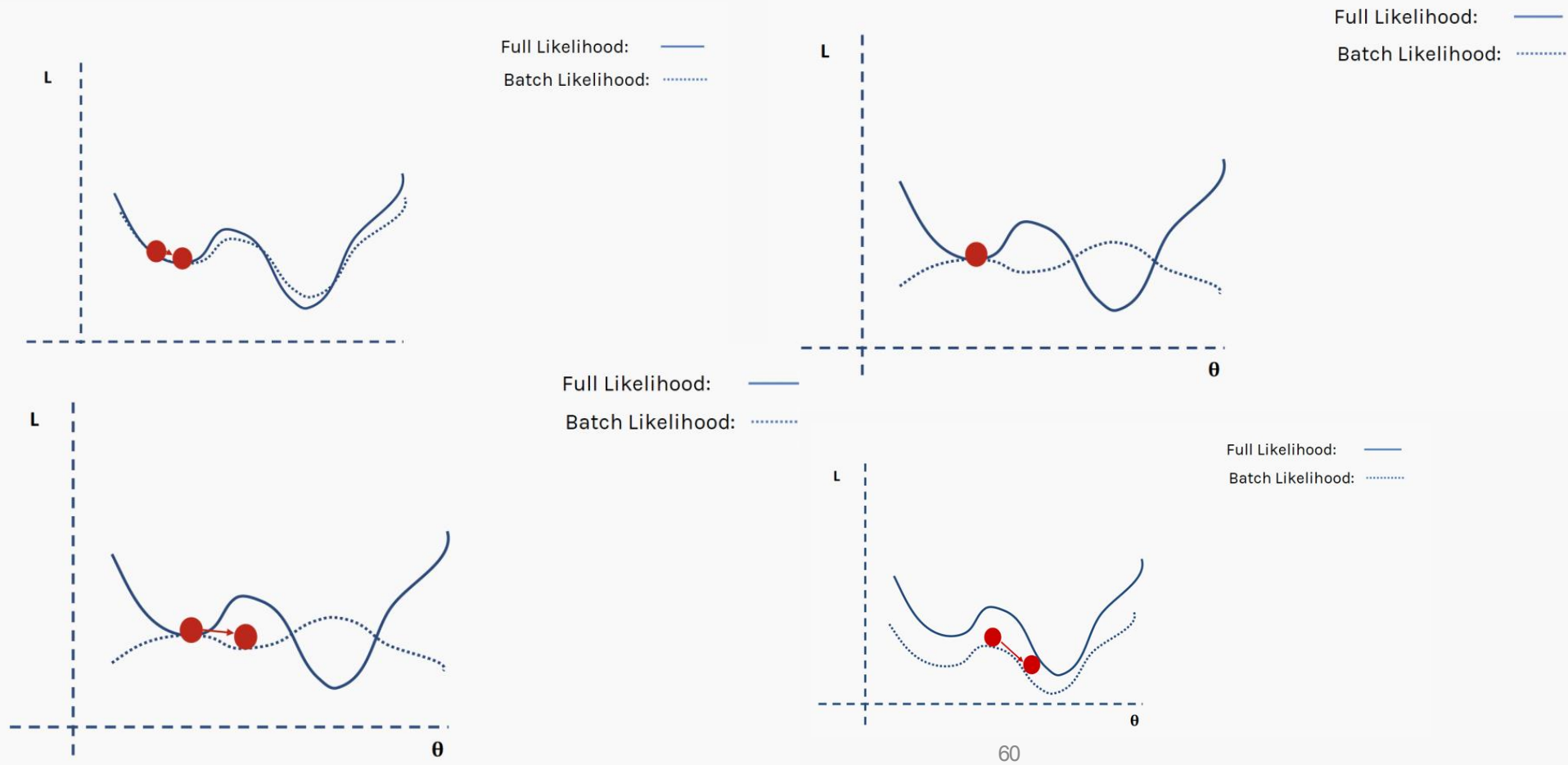
For **stochastic gradient descent**,

Use a randomly selected sample to update the weights a sample at a time. (Like batch, but here size = 1)

Batch Gradient Descent



Batch Gradient Descent



MNIST Training

```
def train(model, X_train, y_train, X_valid, y_valid, num_epochs,
          learning_rate=0.1):

    epoch_loss = []
    epoch_train_acc = []
    epoch_valid_acc = []

    for e in range(num_epochs):

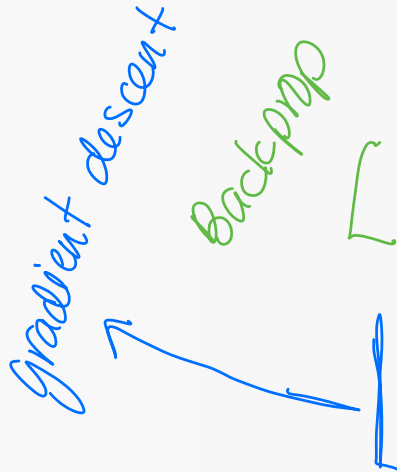
        # iterate over minibatches
        minibatch_gen = minibatch_generator(
            X_train, y_train, minibatch_size)

        for X_train_mini, y_train_mini in minibatch_gen:

            ##### Compute outputs #####
            a_h, a_out = model.forward(X_train_mini)

            ##### Compute gradients #####
            d_loss__d_w_out, d_loss__d_b_out, d_loss__d_w_h, d_loss__d_b_h = \
                model.backward(X_train_mini, a_h, a_out, y_train_mini)

            ##### Update weights #####
            model.weight_h -= learning_rate * d_loss__d_w_h
            model.bias_h -= learning_rate * d_loss__d_b_h
            model.weight_out -= learning_rate * d_loss__d_w_out
            model.bias_out -= learning_rate * d_loss__d_b_out
```

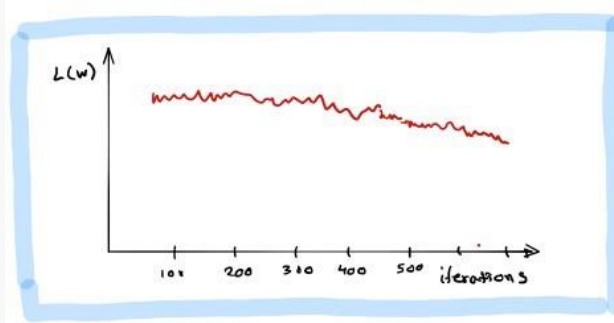
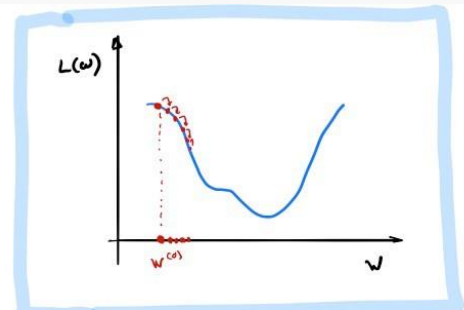


MNIST Training

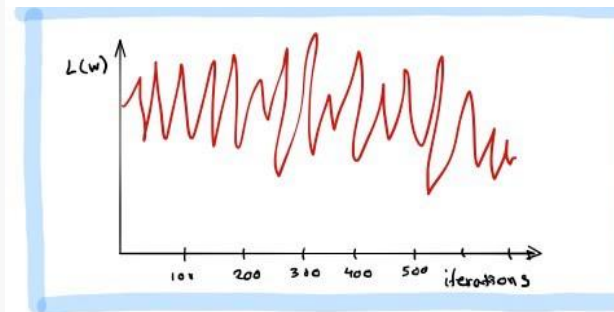
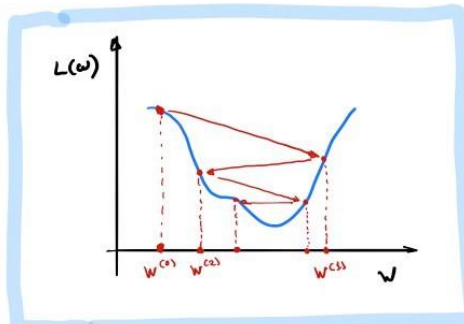
```
##### Epoch Logging #####
train_mse, train_acc = compute_mse_and_acc(model, X_train, y_train)
valid_mse, valid_acc = compute_mse_and_acc(model, X_valid, y_valid)
train_acc, valid_acc = train_acc*100, valid_acc*100
epoch_train_acc.append(train_acc)
epoch_valid_acc.append(valid_acc)
epoch_loss.append(train_mse)
print(f'Epoch: {e+1:03d}/{num_epochs:03d} '
      f'| Train MSE: {train_mse:.2f} '
      f'| Train Acc: {train_acc:.2f}% '
      f'| Valid Acc: {valid_acc:.2f}%')

return epoch_loss, epoch_train_acc, epoch_valid_acc
```

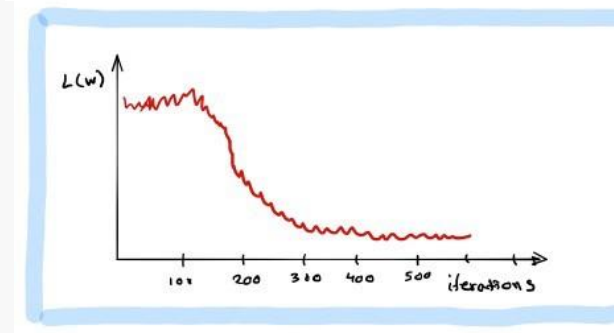
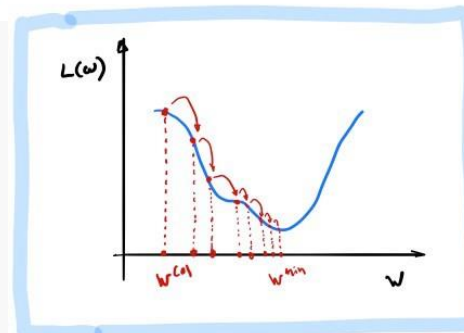

How can we tell when gradient descent is converging? We visualize the loss function at each step of gradient descent. This is called the **trace plot**.



While the loss is decreasing throughout training, it does not look like descent hit the bottom.

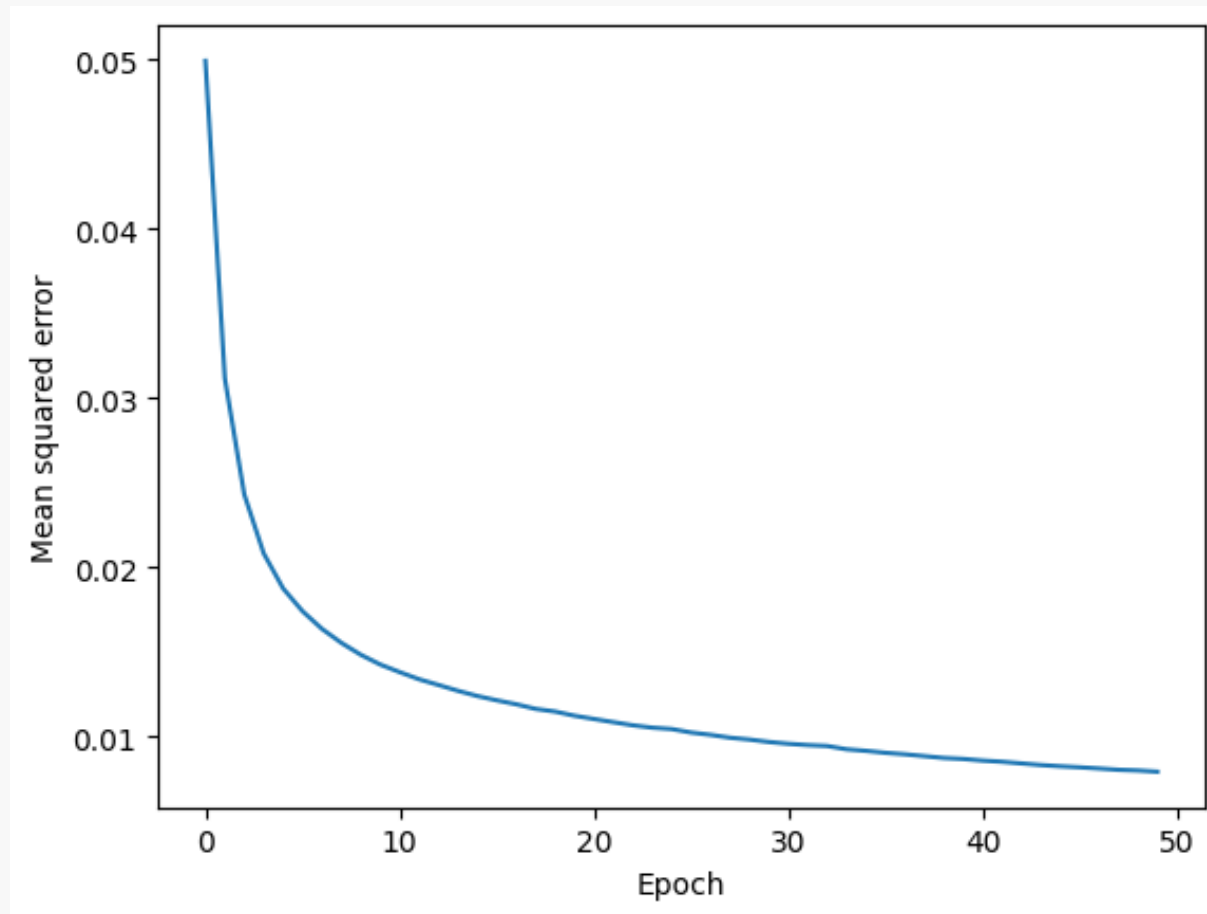


Loss is mostly oscillating between values rather than converging.



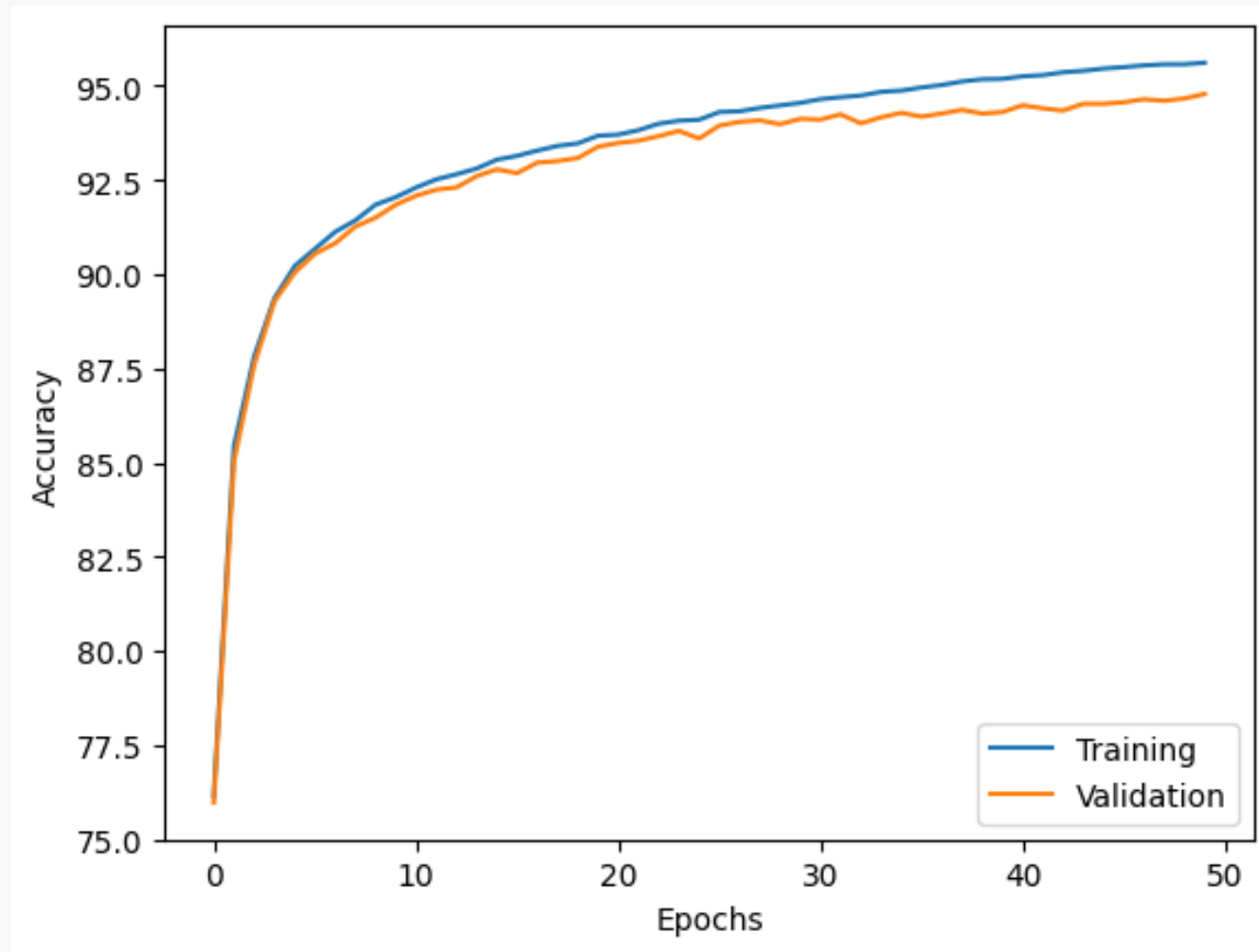
The loss has decreased significantly during training. Towards the end, the loss stabilizes and it can't decrease further.

MNIST Training



[Raschka et al 2022]

MNIST Training



[Raschka et al 2022]

How to get best learning rate

Optimal learning rate = $\frac{1}{2}$ maximum learning rate
(rate above which training diverges)

Approach: Train for hundreds of iterations with low learning rate increasing to large learning rate

1. Start with learning rate = 10^{-5}
2. Multiply learning rate by constant to get learning rate to 10 by end of number of iterations
3. Plot loss as function of learning rate. It should drop, but then when too high, loss will increase

60

How to get best learning rate

Optimal learning rate = $\frac{1}{2}$ maximum learning rate
(rate above which training diverges)

Optimal learning rate will be less than rate at which loss increased again.

Re-train model using that rate.

Pytorch has learning rate schedulers such as
LinearLR

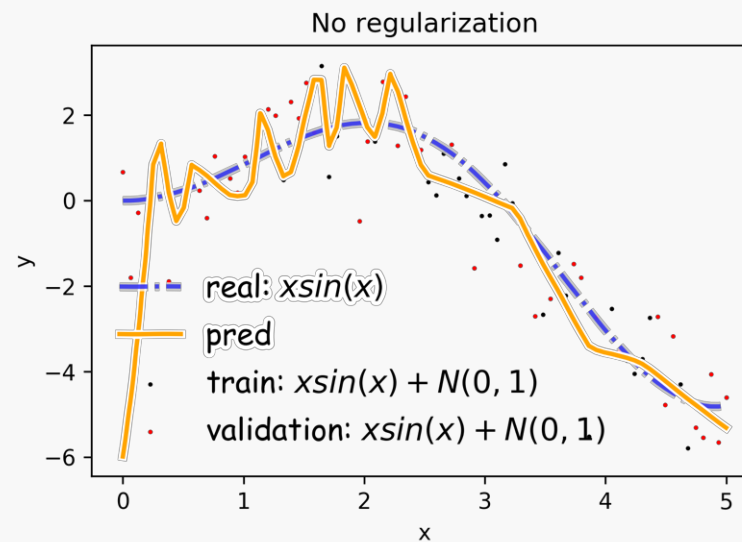
Today's Learning Objectives

Students will be able to:

- ✓ Review: Backprop on NN with MNIST Code
- ✓ Review: Understand the role of gradient descent in training NN with MNIST Code
 - Regularization for NN

Overfitting

Fitting a deep neural network with 5 layers and 100 neurons per layer can lead to a very good prediction on the training set but poor prediction on validation set.



12
0

What is regularization?

Regularization is any modification we make to a learning algorithm that is intended to *reduce its generalization error* but not its training error.

Regularization of NN

1. Norm penalties
2. Early Stopping
3. Dropout

Ridge Regression

- Add a **regularization penalty** to the loss function
- Loss artificially increases as the values of b_0 and b_1 move farther from a particular point such as $b_0=0$ and $b_1=0$.
- Since LS algorithm minimizes loss, this forces b_0 and b_1 to stay relatively close to $b_0=0$ and $b_1=0$.
- Quadratic (squared) L2 penalty term is called **L2 regularization**
- **Ridge regression algorithm** is L2 penalty term with Least Squares algorithm

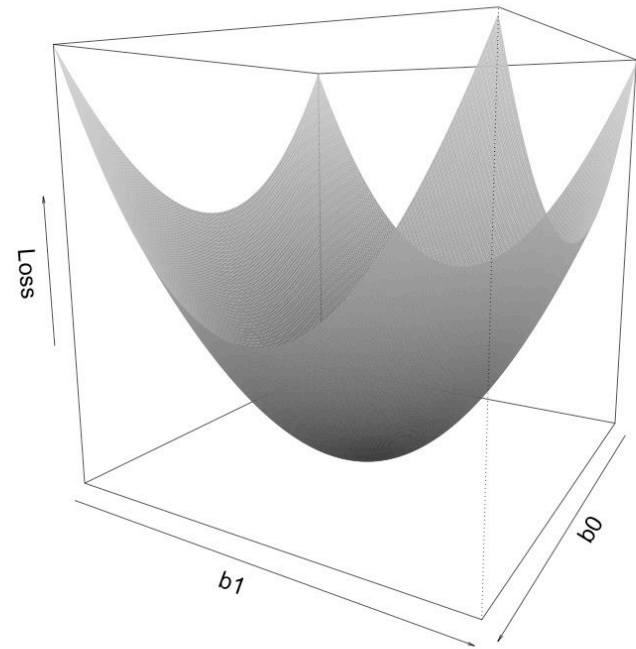
Ridge Regression

Find the values of b_0 and b_1 that make the regularized LS loss

$$\sum_i (\text{observed price}_i - (b_0 + b_1 \text{area}_i))^2 + \lambda(b_0^2 + b_1^2)$$

as small as possible (for some $\lambda \geq 0$).

- Quadratic (squared) L2 **penalty term** is called **L2 regularization**
- **Regularization hyperparameter is λ**



Lasso Regression

- Add a **regularization penalty** to the loss function
- Absolute value or L1 penalty term is called **L1 regularization**
- **Lasso regression algorithm** is L1 penalty term with Least Squares algorithm

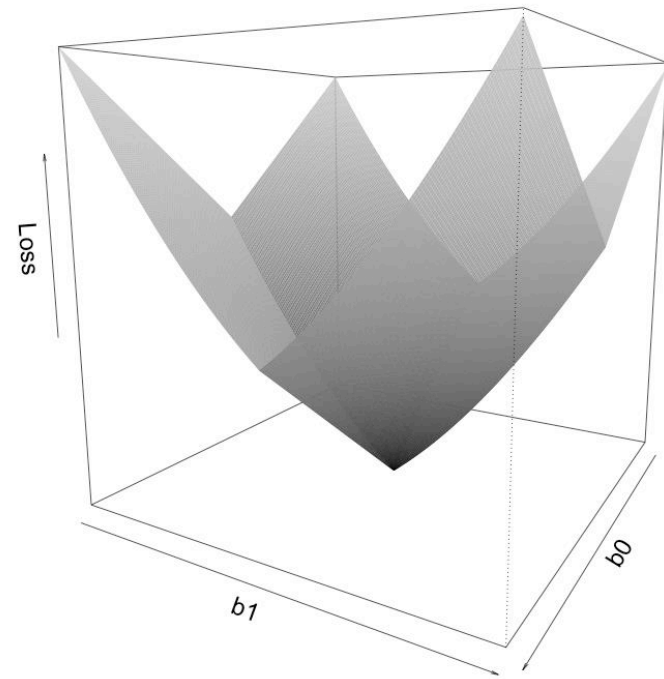
Lasso Regression

Find the values of b_0 and b_1 that make the regularized LS loss

$$\sum_i (\text{observed price}_i - (b_0 + b_1 \text{area}_i))^2 + \lambda(|b_0| + |b_1|)$$

as small as possible (for some $\lambda \geq 0$).

- Absolute value or L1 **penalty term** is called **L1 regularization**
- **Regularization hyperparameter is λ**



Norm Penalties

We used to optimize:

$$L(W; X, y)$$

α hyper parameter

Change to ...

$$L_R(W; X, y) = L(W; X, y) + \alpha \Omega(W)$$

Biases not
penalized

norm
penalty

L_2 regularization:

- Weights decay

$$\Omega(W) = \frac{1}{2} \| W \|_2^2$$

L_1 regularization:

- encourages sparsity

$$\Omega(W) = \frac{1}{2} \| W \|_1$$

Norm Penalties

We used to optimize:

$$W^{(i+1)} = W^{(i)} - \lambda \frac{\partial L}{\partial W}$$

Change to:

$$L_R(W; X, y) = L(W; X, y) + \frac{1}{2} \alpha \|W\|_2^2$$

$$W^{(i+1)} = W^{(i)} - \lambda \frac{\partial L}{\partial W} - \lambda \alpha W^{(i)}$$

Weights decay
in proportion
to size

Biases not
penalized

L_2 regularization:

- Weights decay

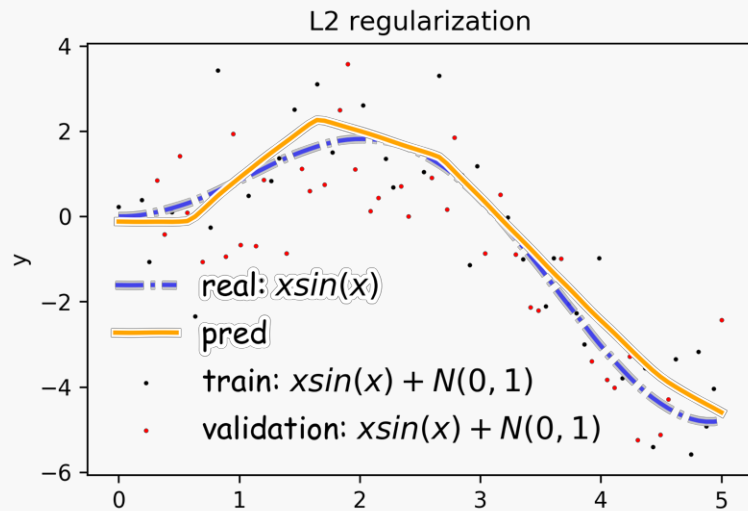
$$\Omega(W) = \frac{1}{2} \|W\|_2^2$$

L_1 regularization:

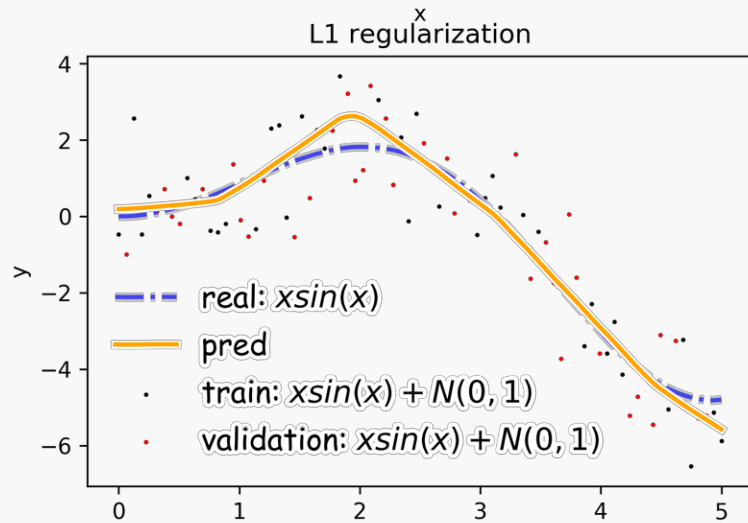
- encourages sparsity

$$\Omega(W) = \frac{1}{2} \|W\|_1$$

Norm Penalties



$$\Omega(W) = \frac{1}{2} \|W\|_2^2$$

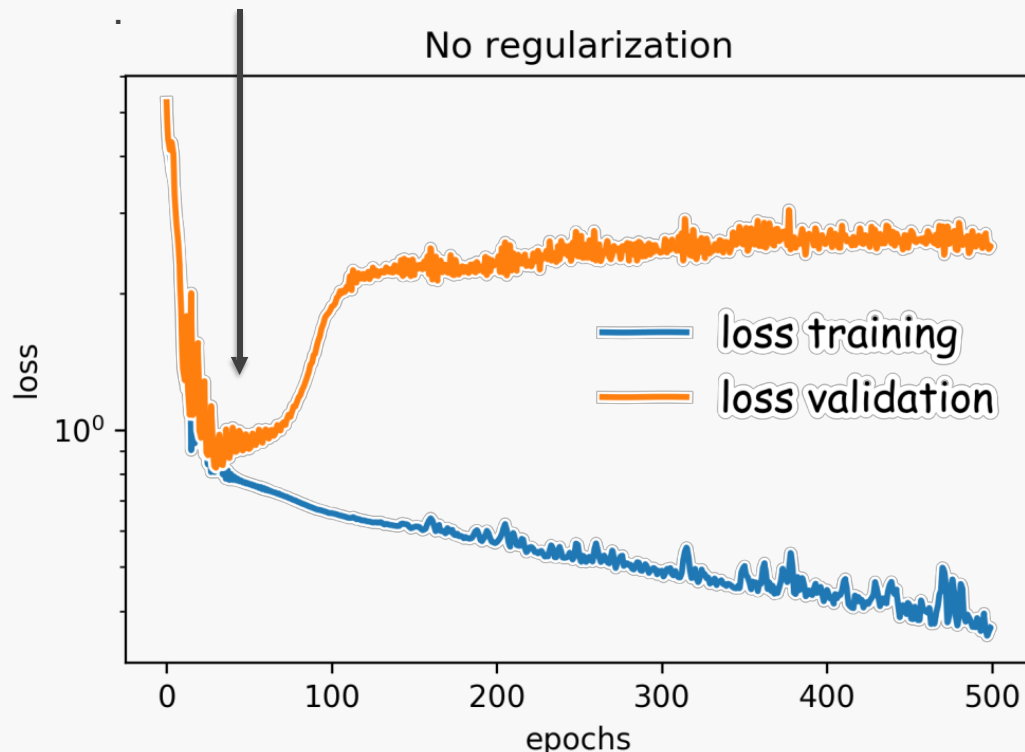


$$\Omega(W) = \frac{1}{2} \|W\|_1$$

Early Stopping

Early stopping: terminate while validation set performance is better. Stop training when validation error reaches a minimum

Hinton calls this a “beautiful free lunch”



Patience is defined as the number of epochs to wait before early stop if no progress on the validation set.

The patience is often set somewhere between **10 and 100** (10 or 20 is more common), but it really depends on the dataset and network.

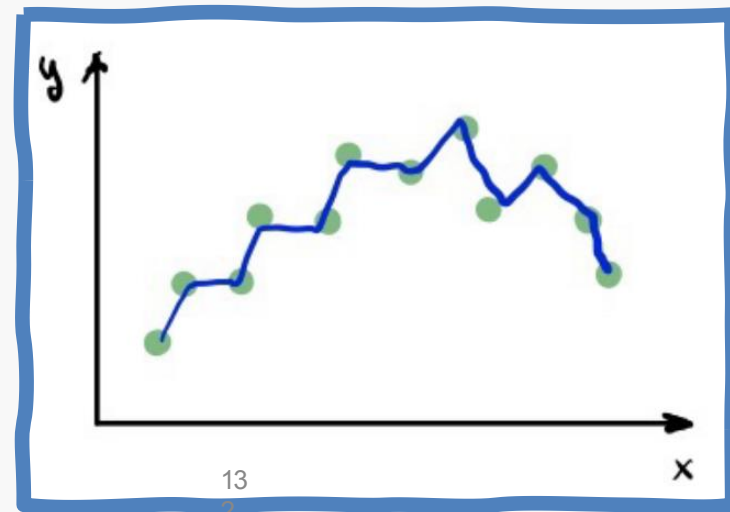
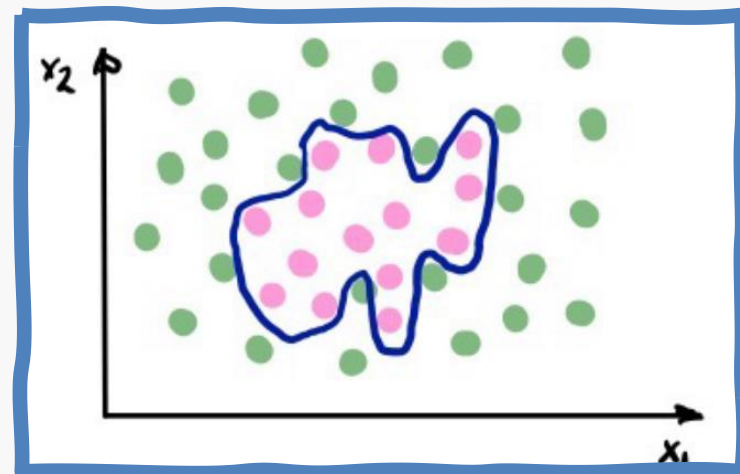
12
5
[CS M148 Winter 2024]

Co-adaptation

Overfitting occurs when the model is **sensitive** to slight variations on the input and therefore it fits the noise.

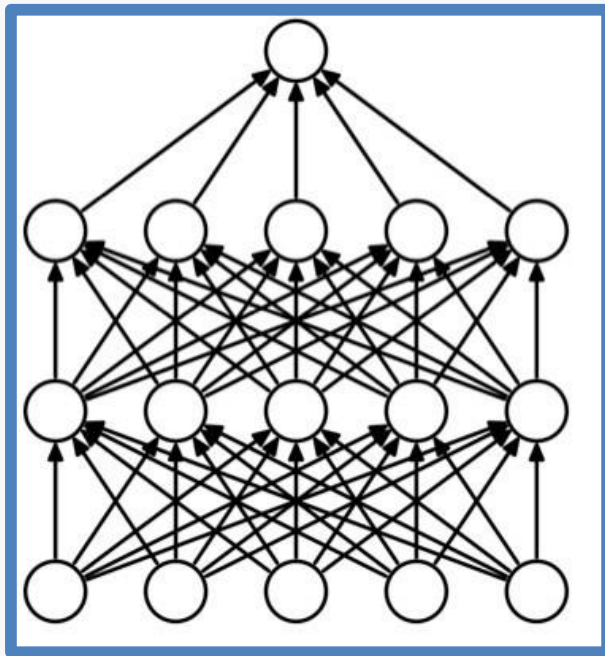
L1 and L2 regularizations ‘shrink’ the weights to avoid this problem.

However in a large network many units can **collaborate** to respond to the input while the weights can **remain relatively small**. This is called **co-adaptation**.

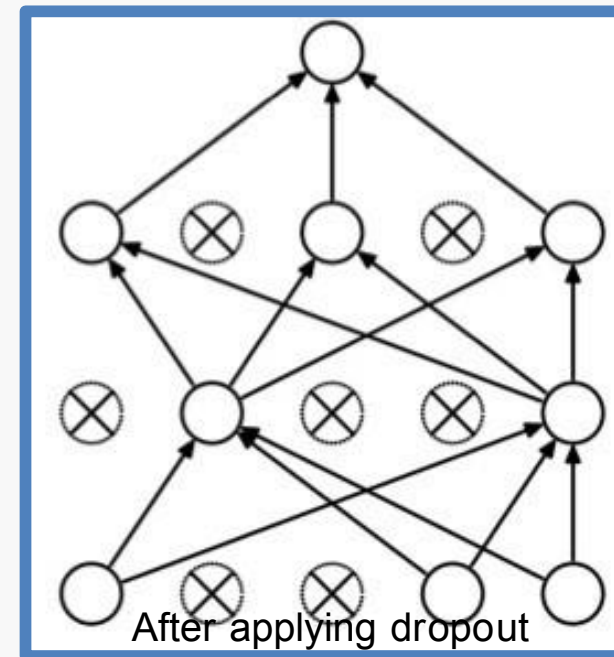


Dropout

- Proposed by Hinton (2012) and Srivastava et al (2014)
- Randomly set some neurons and their connections to zero (i.e. “dropped”)
- Prevent overfitting by reducing **co-adaptation** of neurons
- Like training many random sub-networks



Standard Neural Network



<https://arxiv.org/pdf/1207.0580.pdf>

Dropout: Training

For each new example in a mini-batch (could be for one mini-batch depending on the implementation):

- Randomly **sample a binary mask** μ independently, where μ_i indicates if input/hidden node i is included
- **Multiply output of node i with μ_i** , and perform gradient update

Typically:

- **Input** nodes are included with **prob=0.8** (as per original paper, but rarely used)
- **Hidden** nodes are included with **prob=0.5**

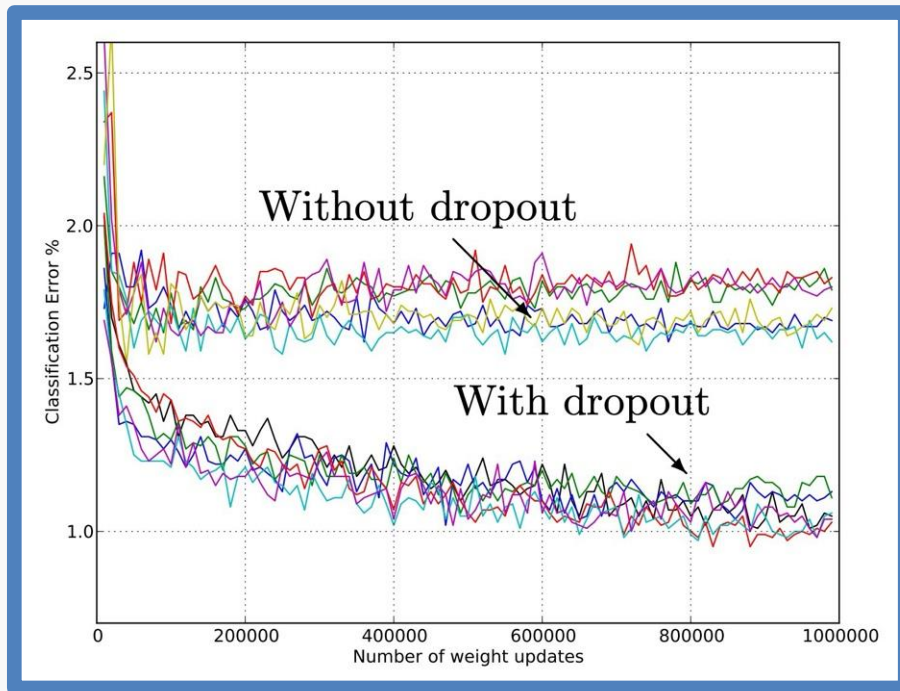
Dropout: Training

During NN training,

- A fraction of hidden neurons dropped at each iteration
- For each hidden neuron, let probability of dropping neuron, $p_{\text{drop}} = .5$ and probability of keeping neuron, $p_{\text{drop}} = 1 - p_{\text{keep}}$
- When input neurons randomly dropped, weights rescaled to account for dropped neurons
- *• Neurons are dropped during training, but all neurons contribute to pre-activations in next layer. why is this important to prevent over-fitting?
- *• Activations are scaled during prediction (using dropout probability) or training (by doubling if $p_{\text{drop}} = .5$)

Dropout

- Widely used and highly effective



Test error for different architectures with and without dropout.

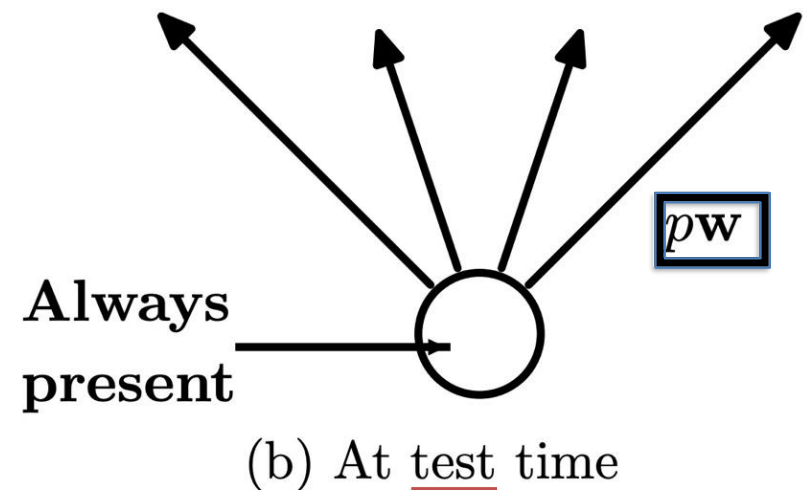
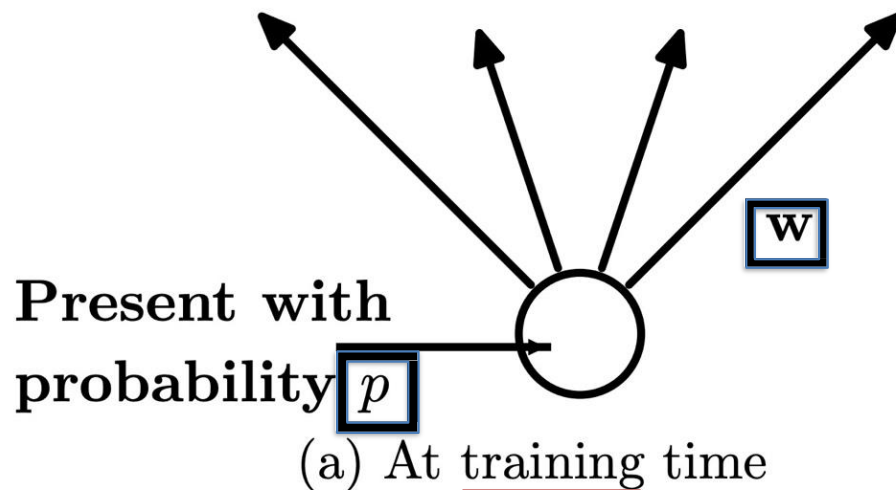
The networks have 2 to 4 hidden layers each with 1024 to 2048 units.

- Proposed as an alternative to ensemble methods, which is too expensive for neural nets

<http://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf>

Dropout: Prediction

- We can think of dropout as training many of sub-networks
- At **test time**, we can “**aggregate**” over these sub-networks by **reducing connection weights in proportion to dropout probability, p**



NOTE: Dropouts can be used for **neural network inference** by dropping during predictions and predicting multiple times to get a distribution

Dropout and Ensemble models

- Proposed as an alternative to [ensemble methods](#), which is too expensive for neural nets

- How are we training many models at once using drop out?

each training iteration; dropping out a subset of nodes is training a different model

- From ^{how} many models are we training?

if there are M neurons in fully connected NN, sampling from 2^M models

Dropout and Ensemble models

- Proposed as an alternative to [ensemble methods](#), which is too expensive for neural nets

- Key difference between drop out and other ensemble methods?

Dropout actually shares weights across all models at once..

- How are we averaging over the models that we are training efficiently?

Next time...

Today's Learning Objectives

Students will be able to:

- ✓ Review: Backprop on NN with MNIST Code
- ✓ Review: Understand the role of gradient descent in training NN with MNIST Code
 - Regularization for NN

Citations:.

Goodfellow, I., Bengio, Y., Courville, A. (2016). *Deep Learning*. MIT Press.

Sebastian **Raschka**, Yuxi (Hayden) **Liu**, and Vahid Mirjalili. **Machine Learning** with PyTorch and Scikit-Learn. Packt Publishing, **2022**.

Baharan Mirzasoileiman, UCLA CS M148 Winter 2024 Lecture 13 Notes

Thank You
