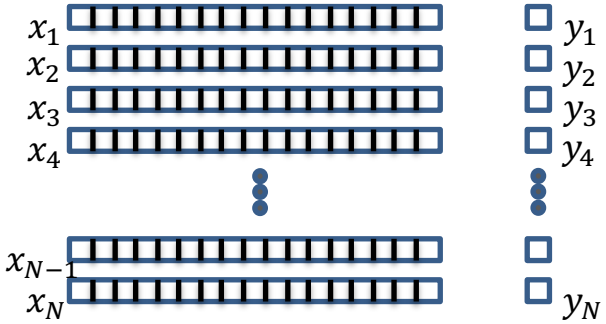
Machine Learning Summer School: Day 1

David Carlson

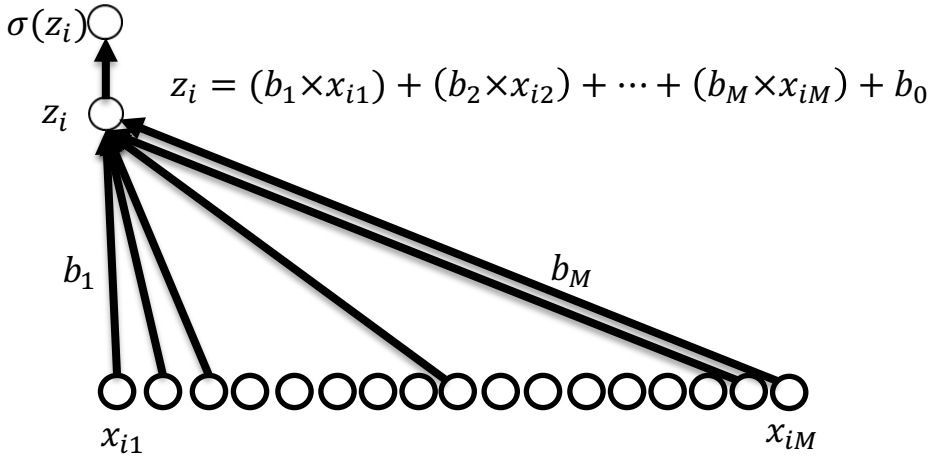
How do we learn a Deep Model?

Part II

(Recall) Learning Model Parameters



Training Set



Logistic Regression Model (or "Network")

How do we actually do this?

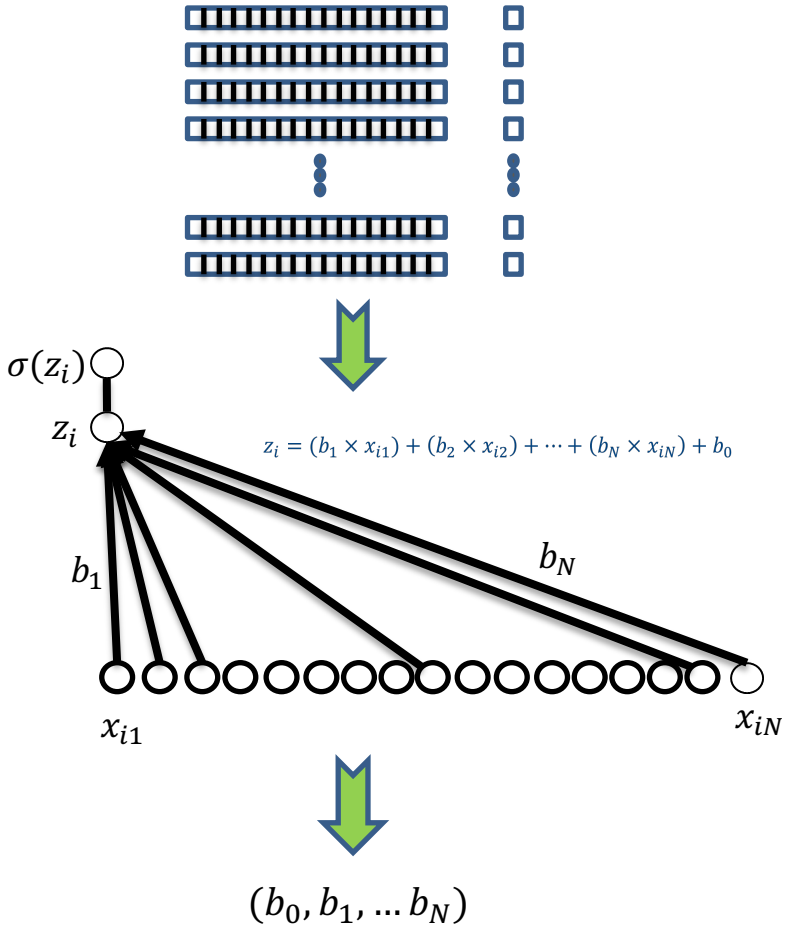
Learned Parameters $(b_0, b_1, \ldots b_N)$

Looking Forward

- This afternoon you can implement and learn a multilayer perceptron *yourself* (with help from the TAs!)
- In this second lecture, we want to set up the mathematical structure to learn the network

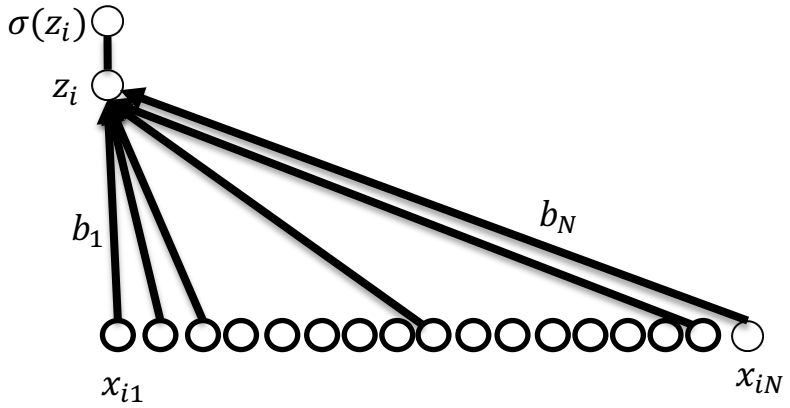
Fundamental Question:

- Given a large amount of data and a model/network to fit, how to *efficiently* and *effectively* learn the model parameters?



What are we trying to do?

- Want to learn parameters that give us the best performance
- Essentially: given data, find the “best” $\boldsymbol{b}$ for that data
- How do we define performance?



“Empirical Risk Minimization”

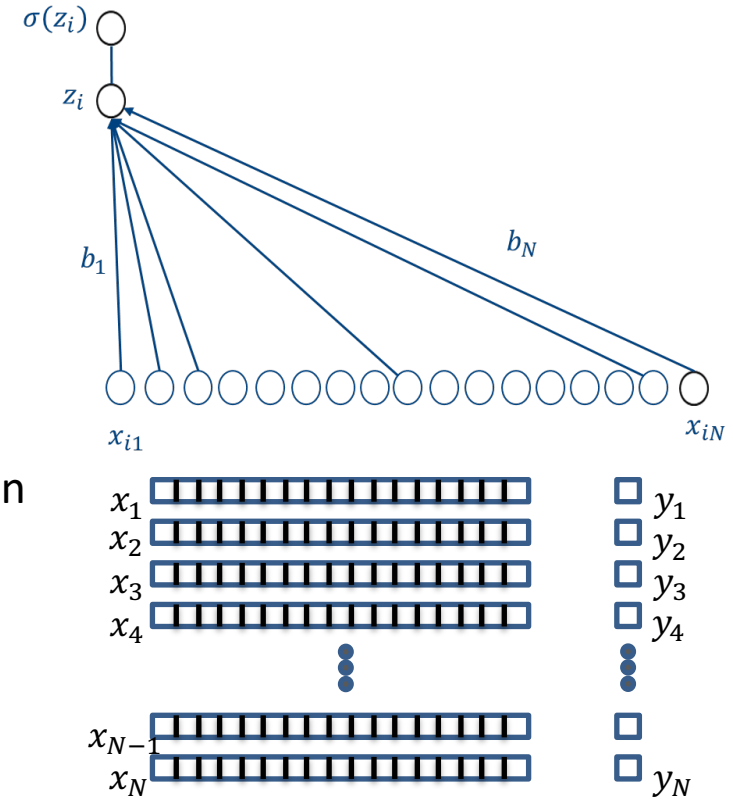
- A *loss function* $\ell(true, prediction)$ defines a penalty for poor predictions
- Want to minimize average loss
- Mathematically, this can be stated as

$$\boldsymbol{b}^* = \arg\min_{\boldsymbol{b}} \frac{1}{N} \sum_{i}^{N} \ell(y_i, \sigma(z_i))$$

Loss function

Guess

True Label



What is the Loss Function?

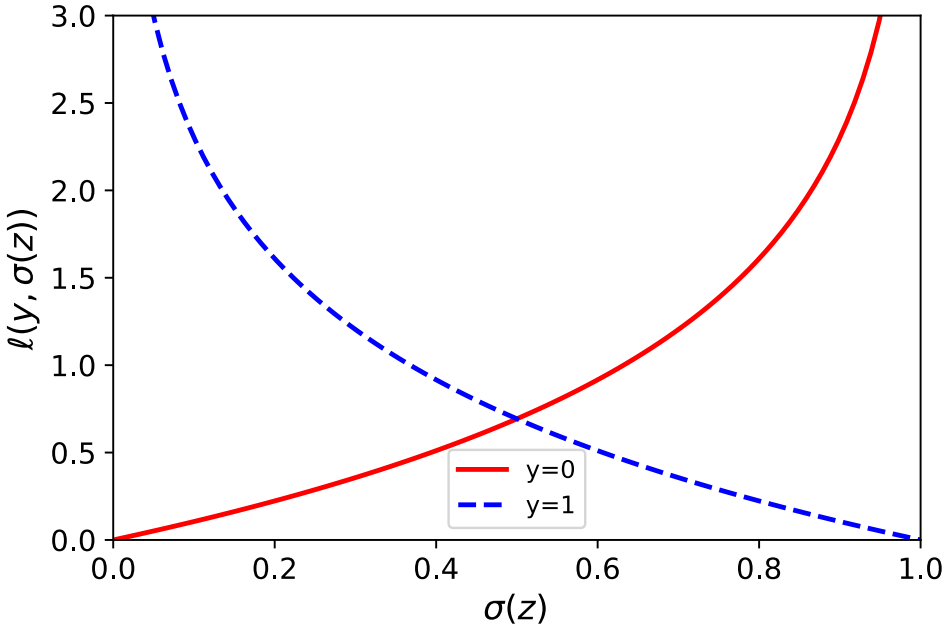
- Define $\sigma(z_i)$ as the predicted probability
- y_i is our true label
- Can be viewed as the negative log-likelihood
  $\ell\big(y_i, \sigma(z_i)\big) = -\log p(y_i|\sigma(z_i))$
- Specific mathematical form is:
  $\ell\big(y, \sigma(z)\big) = -y \log \sigma(z) - (1-y) \log\big(1-\sigma(z)\big)$

Visualization of Logistic Loss Function

- The logistic/cross-entropy loss is:

$$\ell\big(y, \sigma(z)\big) = -y \log \sigma(z) - (1-y) \log\big(1 - \sigma(z)\big)$$

- Dashed blue line shows loss when the true prediction is positive
- When we give 100% of a 1, we pay no penalty
- If we are less confident, we pay a small penalty
- If we are overconfident and wrong, we pay a large penalty



Predicted Probability of a "one"

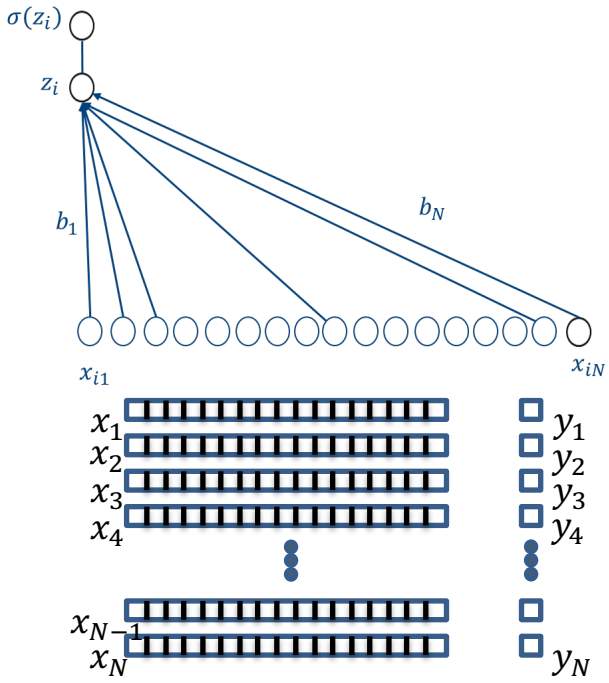
Optimization Goal for Binary Classification

- The optimization goal is to minimize the average loss

$$\boldsymbol{b}^* = \arg\min_{\boldsymbol{b}} \frac{1}{N} \sum_{i}^{N} \ell(y_i, \sigma(z_i))$$

- For binary (0/1) problems, the logistic or cross-entropy loss is

$$\ell\big(y, \sigma(z)\big) = -y \log \sigma(z) - (1-y) \log\big(1 - \sigma(z)\big)$$

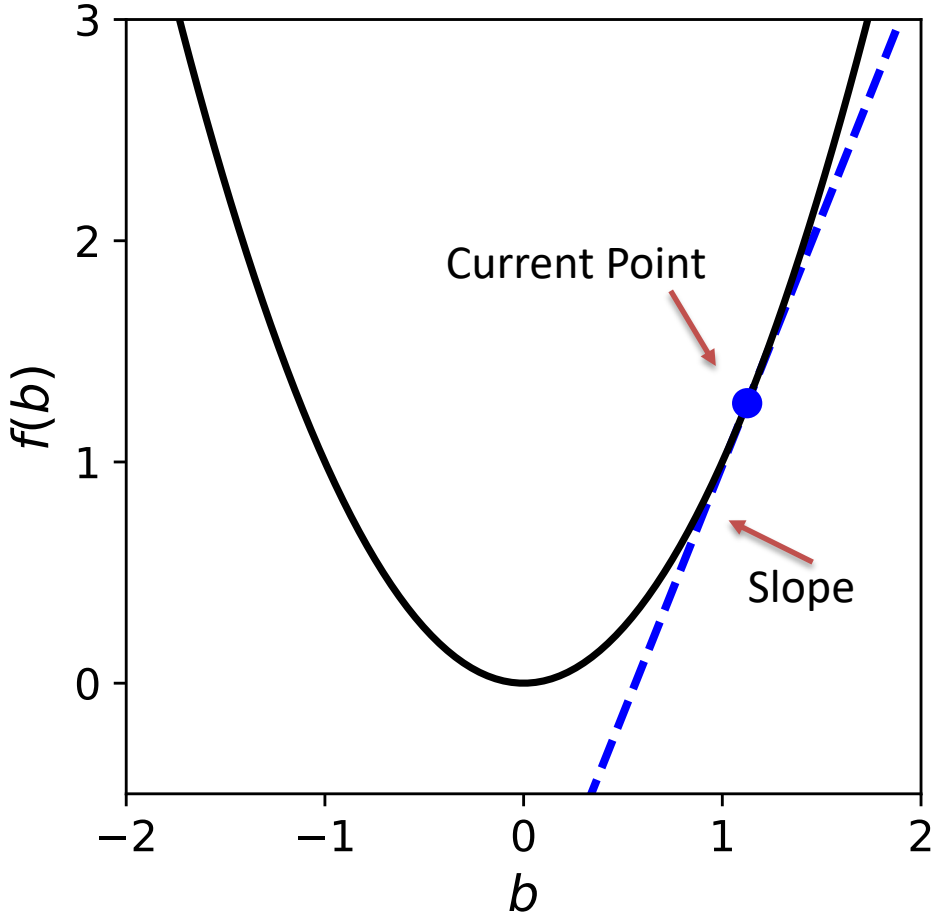


Brief Intro to Gradient Descent

LEARNING THE PARAMETERS

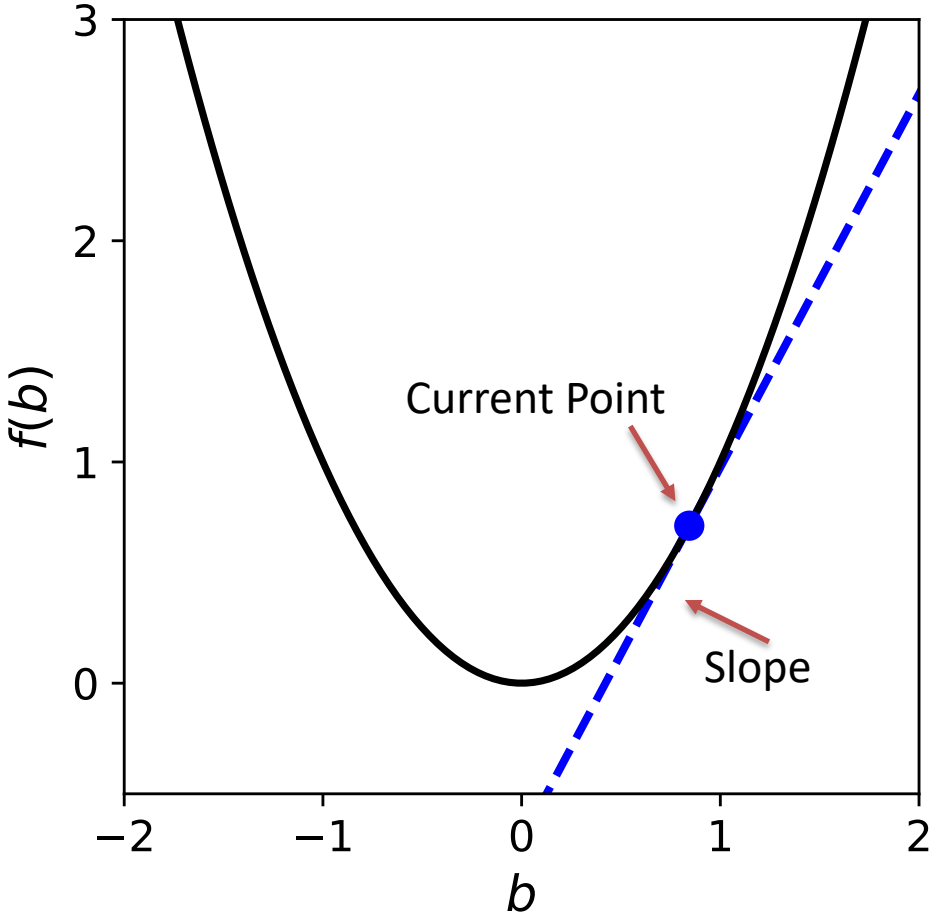
Visualization of Optimization Method

- We want to minimize a mathematical function (i.e. our average loss function)
- One approach is to:
  1. Find the direction pointing “down the hill” (towards a smaller value)
  2. Move a bit in that direction
  3. Repeat 1-2 until satisfied
- This shows the **first** update



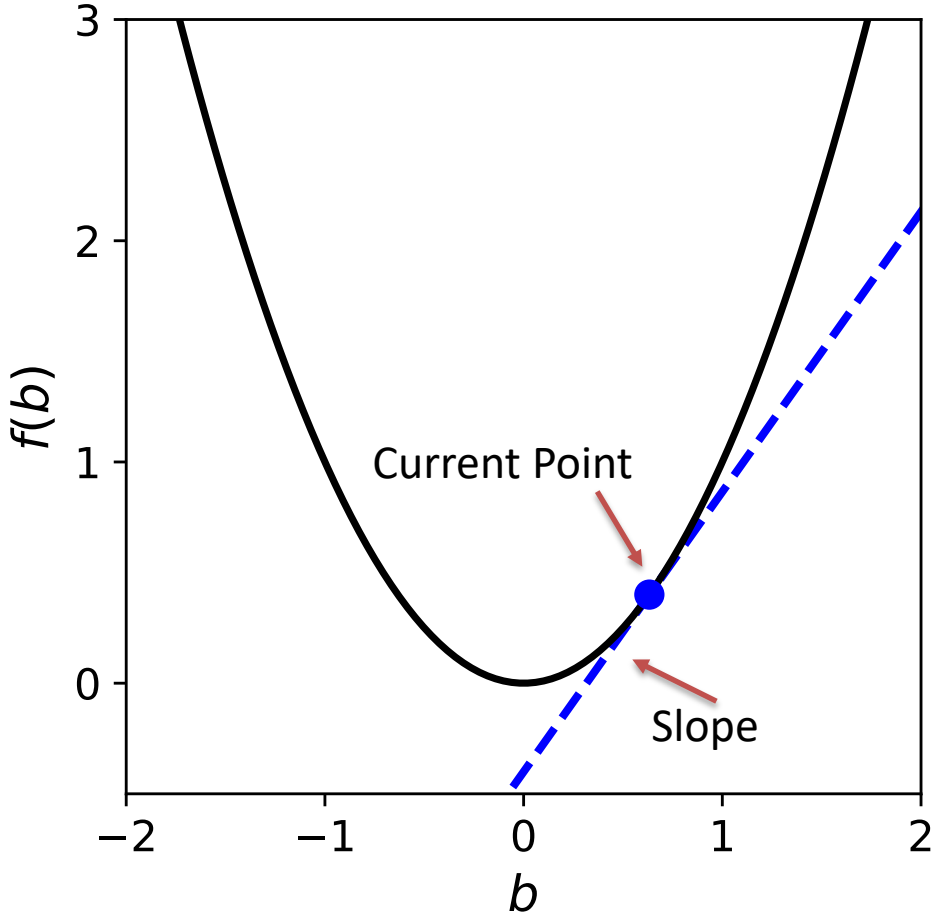
Visualization of Optimization Method

- We want to minimize a mathematical function (i.e. our average loss function)
- One approach is to:
  1. Find the direction pointing “down the hill” (towards a smaller value)
  2. Move a bit in that direction
  3. Repeat 1-2 until satisfied
- This shows the **second** update



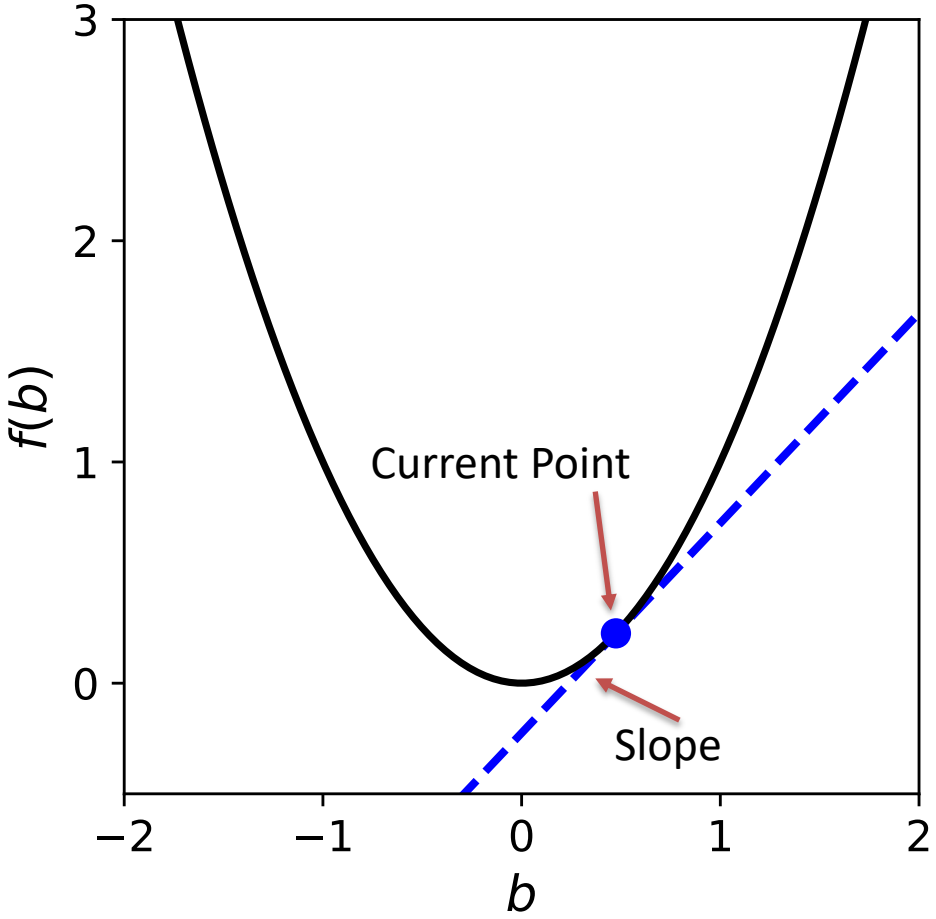
Visualization of Optimization Method

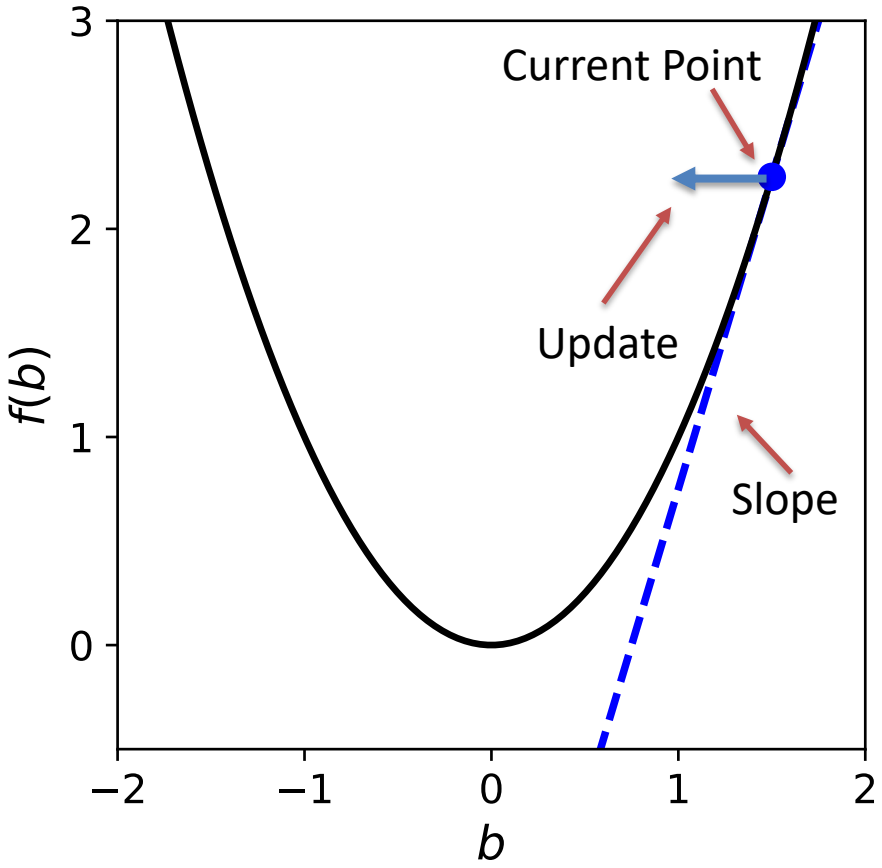
- We want to minimize a mathematical function (i.e. our average loss function)
- One approach is to:
  1. Find the direction pointing “down the hill” (towards a smaller value)
  2. Move a bit in that direction
  3. Repeat 1-2 until satisfied
- This shows the **third** update



Visualization of Optimization Method

- We want to minimize a mathematical function (i.e. our average loss function)
- One approach is to:
  1. Find the direction pointing “down the hill” (towards a smaller value)
  2. Move a bit in that direction
  3. Repeat 1-2 until satisfied
- This shows the **fourth** update





Mathematical Description of Gradient Descent

- We want to minimize a function
$$b^* = \arg\min_b f(b)$$
- Start at an initial value b^0
- We will run a series of updates to move from b^k to b^{k+1} (i.e. from b^0 to b^1)
- Iteratively run the procedure:
  1. Calculate the slope at the current point (For one parameter, this is the derivative. For multiple parameters, this is the *gradient*.): $\nabla f(b^k)$, ∇ means gradient or multidimensional slope
  2. Move in the direction of the negative direction of the gradient with *step size* α^k: $b^{k+1} = b^k - \alpha^k \nabla f(b^k)$
  3. Repeat 1-2 until converged

Brief Introduction to *Stochastic* Gradient Descent (SGD)

SCALING TO BIG DATA

Return to our Optimization Goal

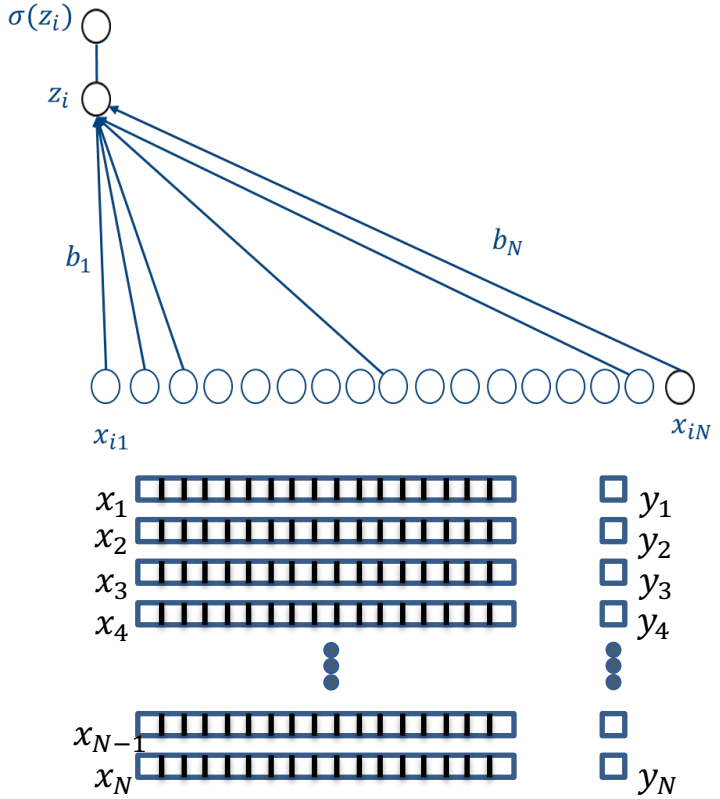
- We want to find the best parameters for all data points

$$\boldsymbol{b}^* = \arg\min_{\boldsymbol{b}} \frac{1}{N} \sum_{i}^{N} \ell(y_i, \sigma(z_i))$$

- However, calculating the gradient requires looking at *every* data point

$$\nabla \frac{1}{N} \sum_{i}^{N} \ell(y_i, \sigma(z_i)) = \frac{1}{N} \sum_{i}^{N} \nabla \ell(y_i, \sigma(z_i))$$

- This can be problematic in big data:
  - MNIST has ~60,000 images
  - ImageNet has ~1,000,000 images



Let's Just Approximate the Gradient

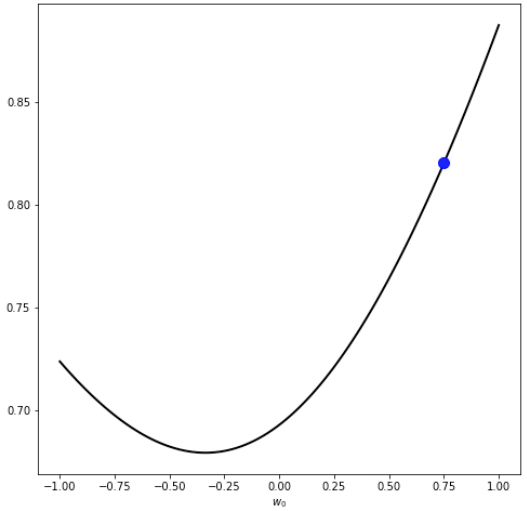
- We don't want to look at *every* data point to update our parameters
- Let's just take a single example j and use it to approximate the gradient$_N$

$$\nabla \ell \left(y_j, \sigma(z_j) \right) \simeq \frac{1}{N} \sum_{i=1}^{N} \nabla \ell (y_i, \sigma(z_i))$$

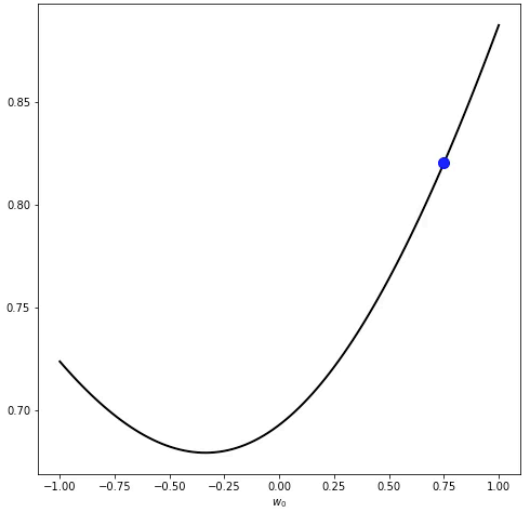
- In ImageNet, this is *1,000,000 times faster*
- Does this work? What would this look like?

Visualizations

Gradient Descent



Stochastic Gradient Descent



Data is Often Redundant

- 60,000 images, but really only 10 different types…

Mathematical Descriptions

Gradient Descent

1. Start with an initial $\boldsymbol{b}^0$
2. Calculate gradient $\nabla f(\boldsymbol{b}^k)$ over *all* data
3. Iteratively update: $\boldsymbol{b}^{k+1} \leftarrow \boldsymbol{b}^k - \alpha_k \nabla f(\boldsymbol{b}^k)$
4. Repeat 2-3 until solution is good enough

Stochastic Gradient Descent

1. Start with an initial $\boldsymbol{b}^0$
2. Choose a random data entry j
3. Estimate gradient $\widehat{\nabla f}\ (\boldsymbol{b}^k)$ by data point j
4. Iteratively update: $\boldsymbol{b}^{k+1} \leftarrow \boldsymbol{b}^k - \alpha_k \widehat{\nabla f}(\boldsymbol{b}^k)$
5. Repeat 2-4 until solution is good enough

Question:

- Reminder that:

$$\mathbb{E}_{j\sim Unif(1,\ldots,\mathrm{N})}\left[\nabla f_j(\boldsymbol{b})\right] = \frac{1}{\mathrm{N}}\sum_{i=1}^{N}\nabla f_i(\boldsymbol{b})$$

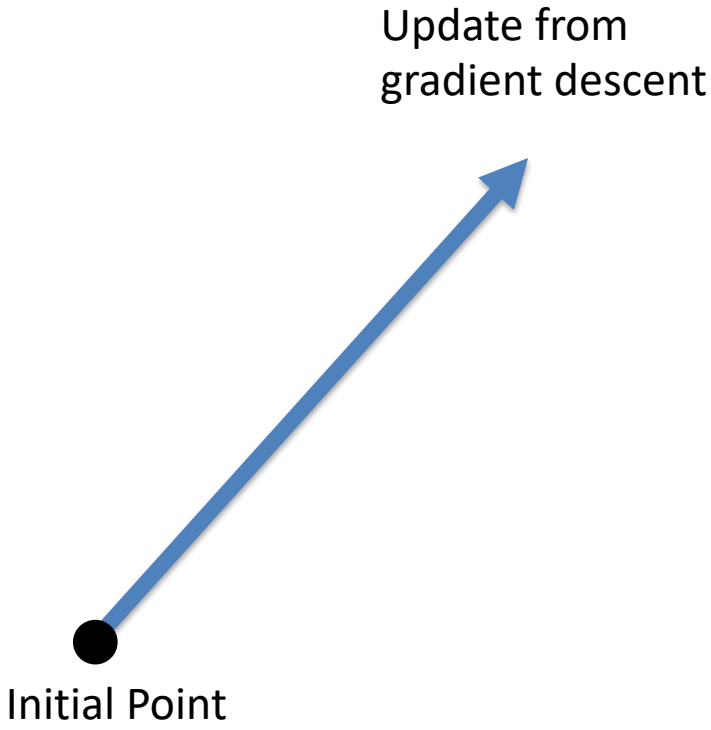
If the updates for gradient descent and stochastic gradient descent start at the same place, how do their resulting updates vary? How about the expectations of their updates?

Our estimation of the gradient is (statistically) unbiased

Our true gradient $\nabla f(\boldsymbol{b}^k)$ is equivalent to the expectation of our stochastic gradient $\nabla f_j(\boldsymbol{b}^k)$.

$$\mathrm{E}_{\mathrm{j}\sim Uniform(1,\ldots,N)}\left[\nabla f_j\left(\boldsymbol{b}^k\right)\right] = \nabla f(\boldsymbol{b}^k)$$

Only difference between gradient descent and stochastic gradient descent is the ***variance*** of the update.

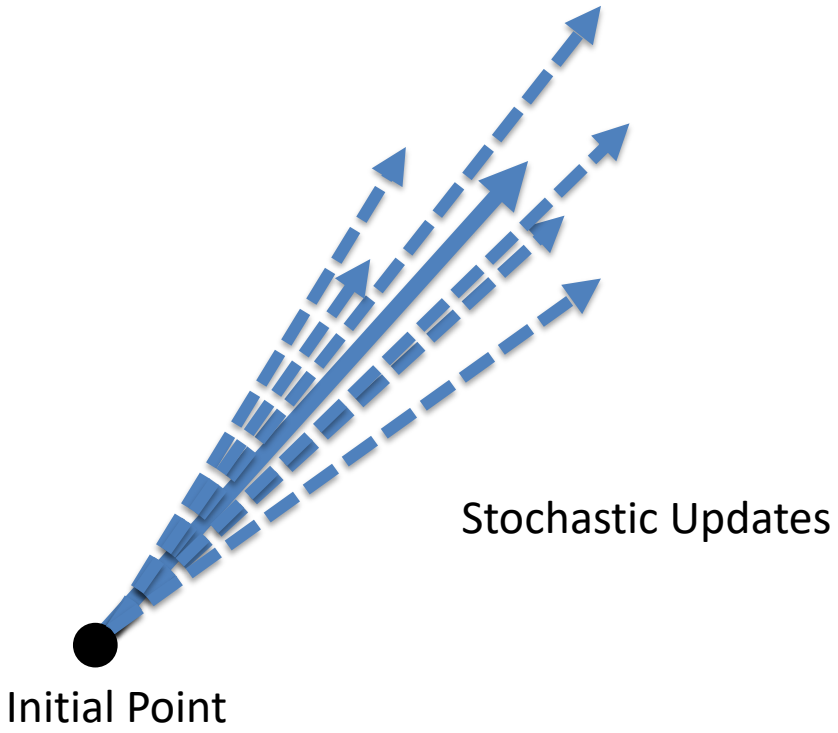


Our estimation of the gradient is (statistically) unbiased

Our true gradient $\nabla f(\boldsymbol{b}^k)$ is equivalent to the expectation of our stochastic gradient $\nabla f_j(\boldsymbol{b}^k)$.

$$\mathrm{E}_{\mathrm{j}\sim Uniform(1,\ldots,N)}\left[\nabla f_j\left(\boldsymbol{b}^k\right)\right] = \nabla f(\boldsymbol{b}^k)$$

Only difference between gradient descent and stochastic gradient descent is the ***variance*** of the update.

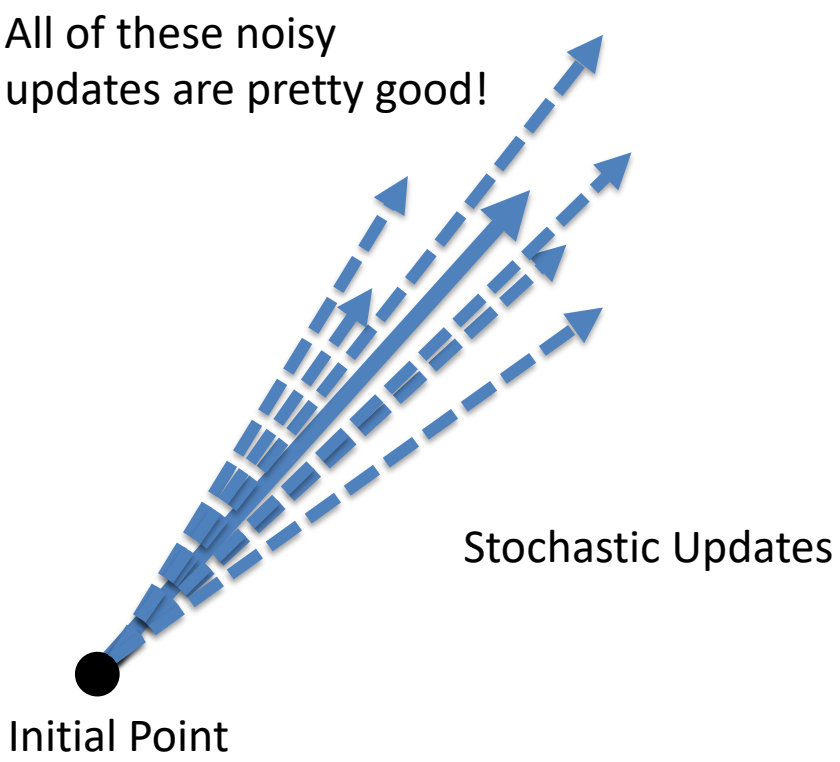


When the magnitude of the gradient is large, all updates help

If a gradient is large, a noisy gradient still points in *mostly* the same direction.

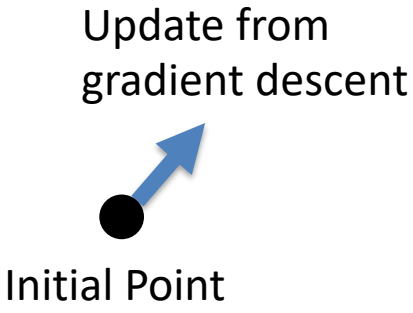
When we are far from the solution, the gradient is usually large. *Our updates are then pretty good when we're far from the solution.*

What happens as we get closer to the solution?



This variance effects smaller updates more

When the magnitude of the gradient is smaller, the noise is more impactful.

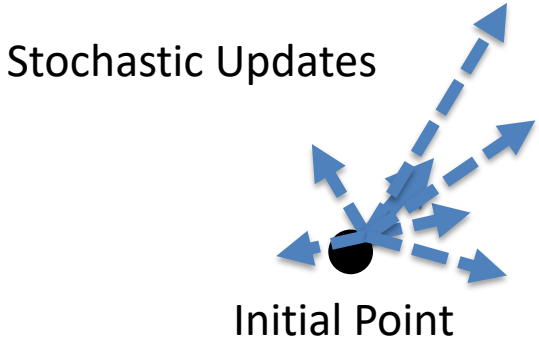


This variance effects smaller updates more

When the magnitude of the gradient is smaller, the noise is more impactful.

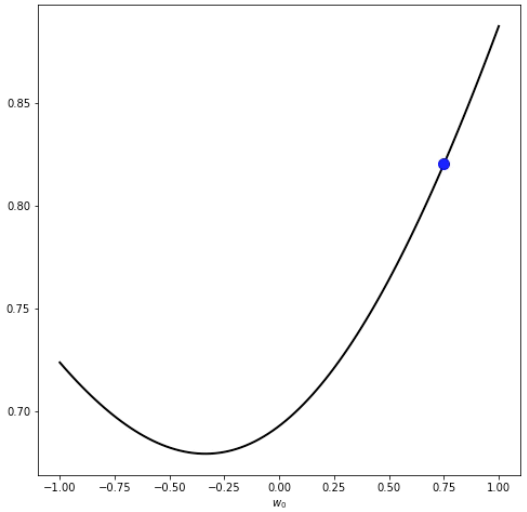
Stochastic updates are more likely to go in the wrong direction in this case.

…but gradients are typically only small when we're near a good solution!

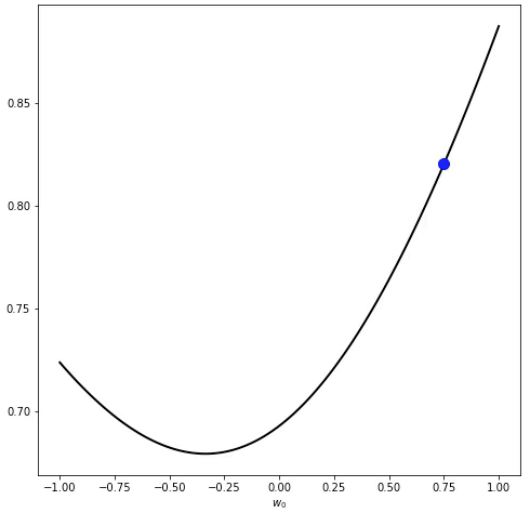


Returning to our Visualization

Gradient Descent



Stochastic Gradient Descent

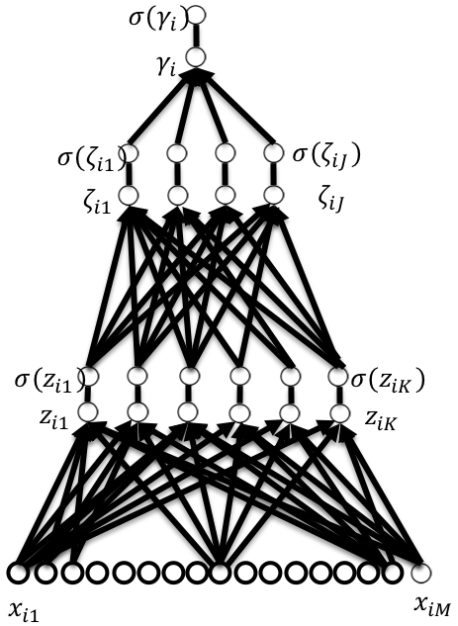


Conclusion on SGD

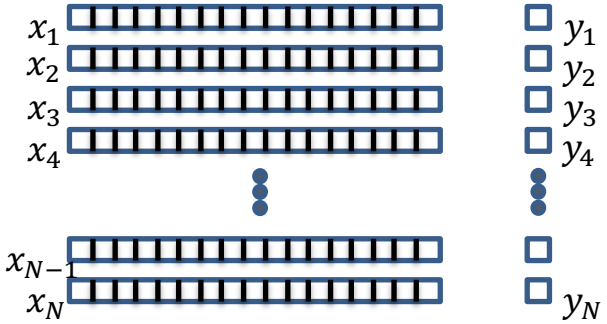
- Stochastic Gradient Descent can update *many more times* than Gradient Descent
- Gets ***near*** the solution very quickly
- Allows scaling to *big data* (update time doesn't increase with the data size)
- Much of the more advanced methods use techniques to minimize the variance (or improve signal-to-noise ratios):
  - Use a "minibatch," which uses a few samples to estimate the gradient
  - SGD with momentum (an algorithmic modification)
  - Smaller step sizes

Calculating the Gradient

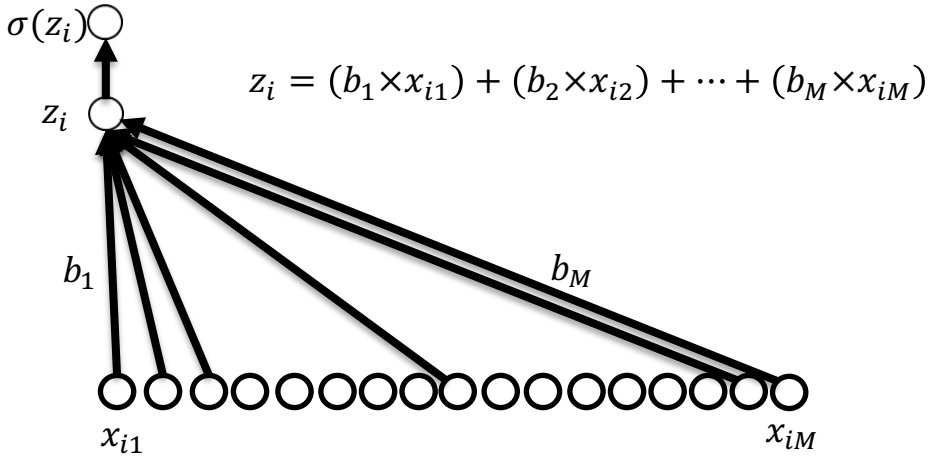
- Calculating the gradient (multi-dimensional slope) on a multi-layer perceptron may seem daunting
- Algorithms can automatically calculate the gradient ("backpropagation") using chain rule from calculus
- In code, we simply have to use a "gradient" function call once the network is defined



Learning Model Parameters (Recap)



Training Set



Logistic Regression Model (or "Network")

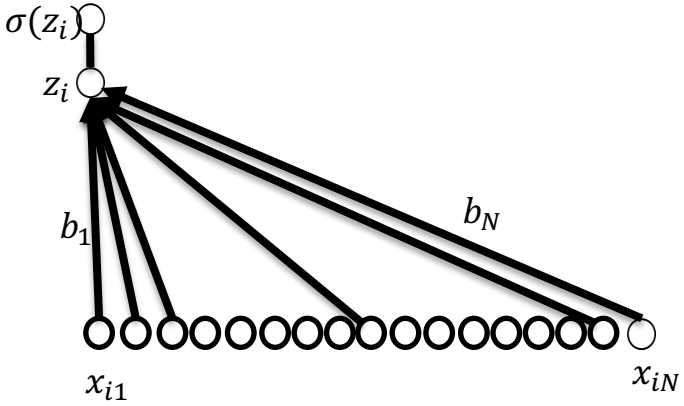
Stochastic Gradient Descent

Learned Parameters $(b_0, b_1, \ldots b_N)$

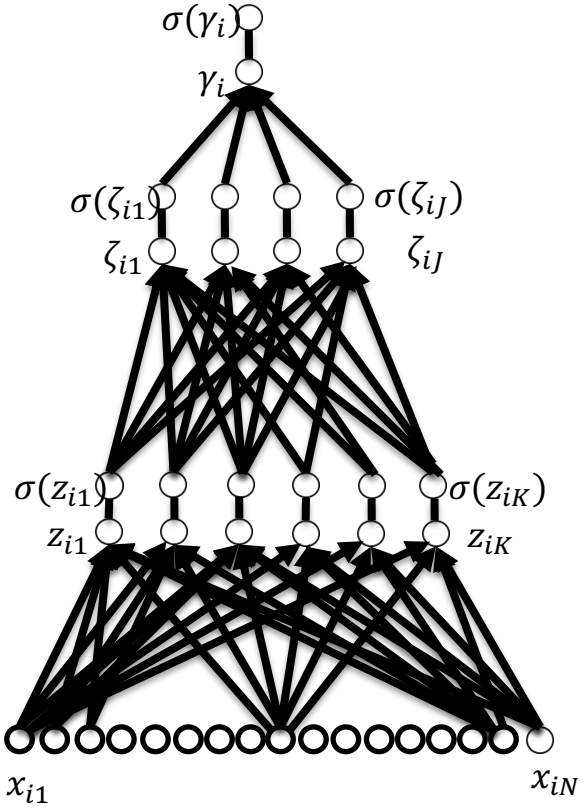
An Introduction to Model Validation

ESTIMATING PERFORMANCE

Do we always want to increase complexity?



?

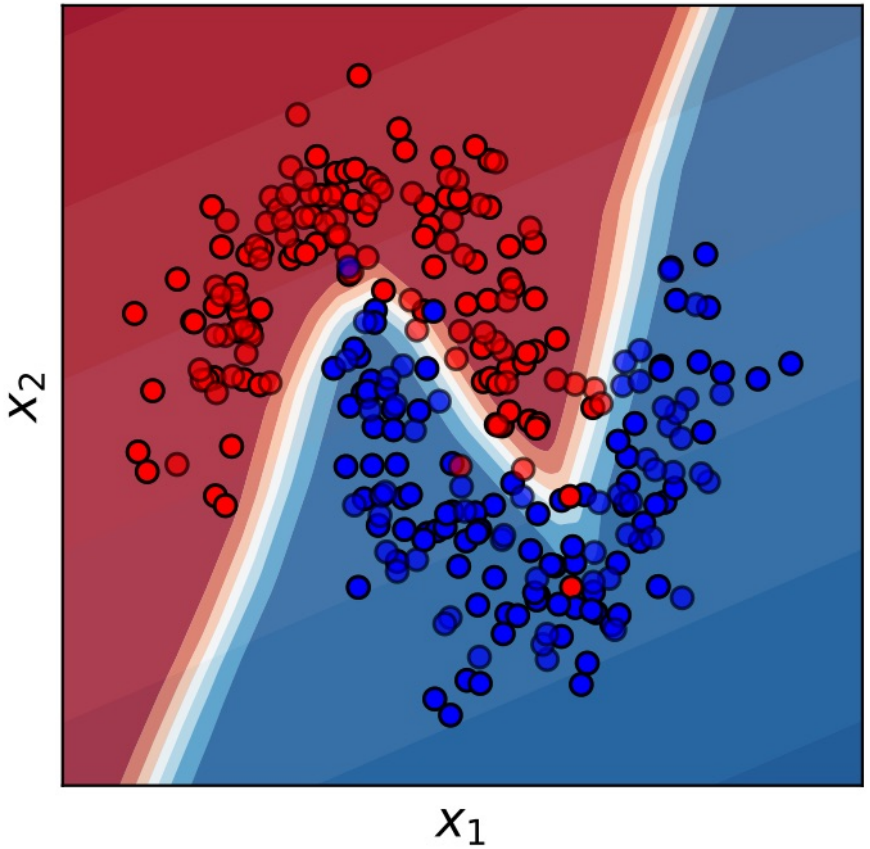


Creating deep models can help us learn complex relationships

My learning multiple layers and transformation, it is possible to have a non-linear classifier capable of more accurately capturing the data.

However, a deep model can also give “perfect” performance on the training dataset and **fail completely** in the real world

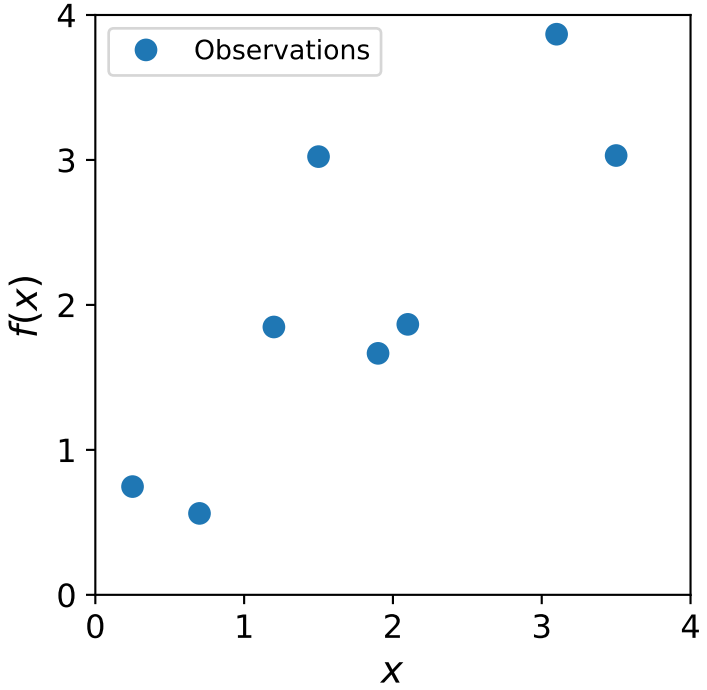
We need to *validate* the performance



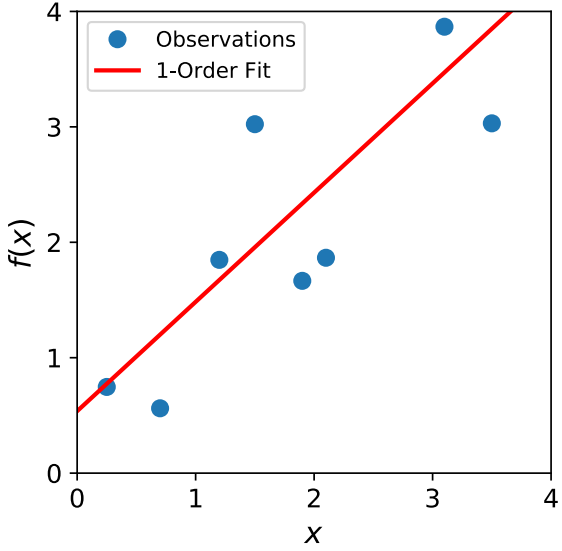
Overfitting

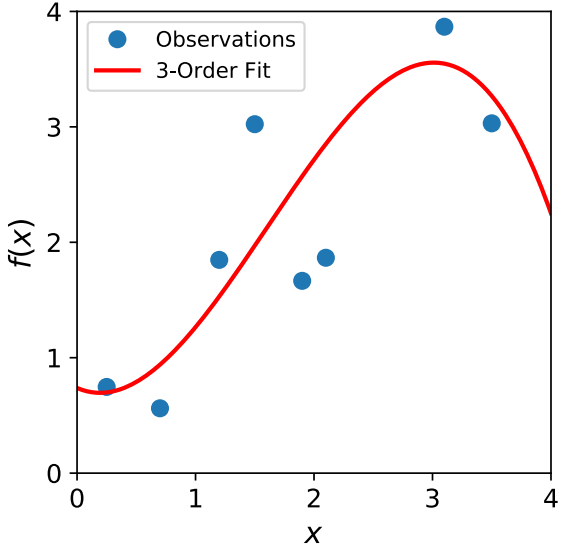
"Overfitting" is when the learned model increases complexity to fit the observed training data *too well* – will not work to predict future data!

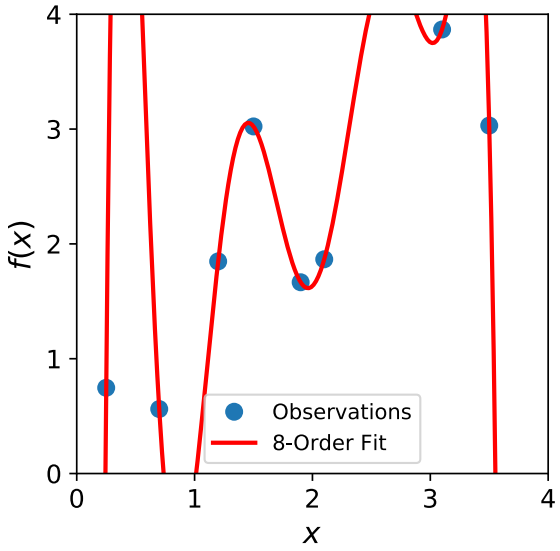
What would we want to use to fit these example data points?



Classic Example: Increasing Polynomial Order





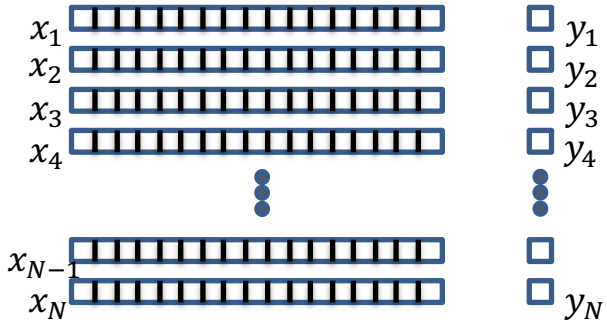


Increasing complexity visually looks ridiculous…

What happens in overfitting?

- We are increasing the number of parameters in the model, which means:
  - More parameters to estimate (all of their errors add up)
  - Can learn *complex* relationships, maybe too complex for reality

- When we *overfit*, this means that we will not *generalize*
  - We want our models and analysis to generalize, or provide accurate predictions on newly collected data



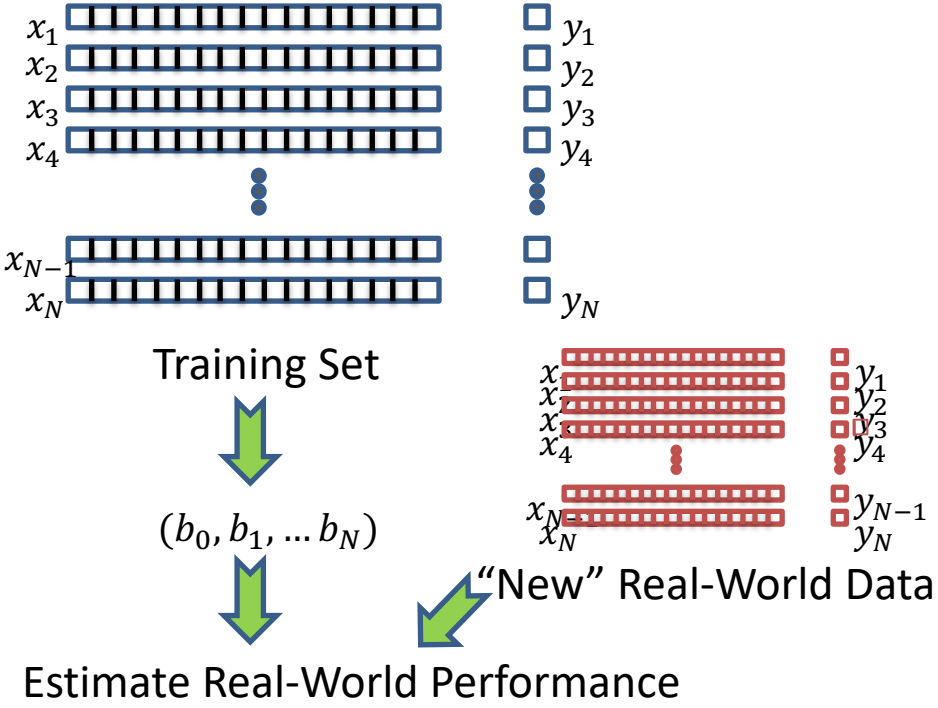
Training Set

$(b_0, b_1, \ldots b_N)$

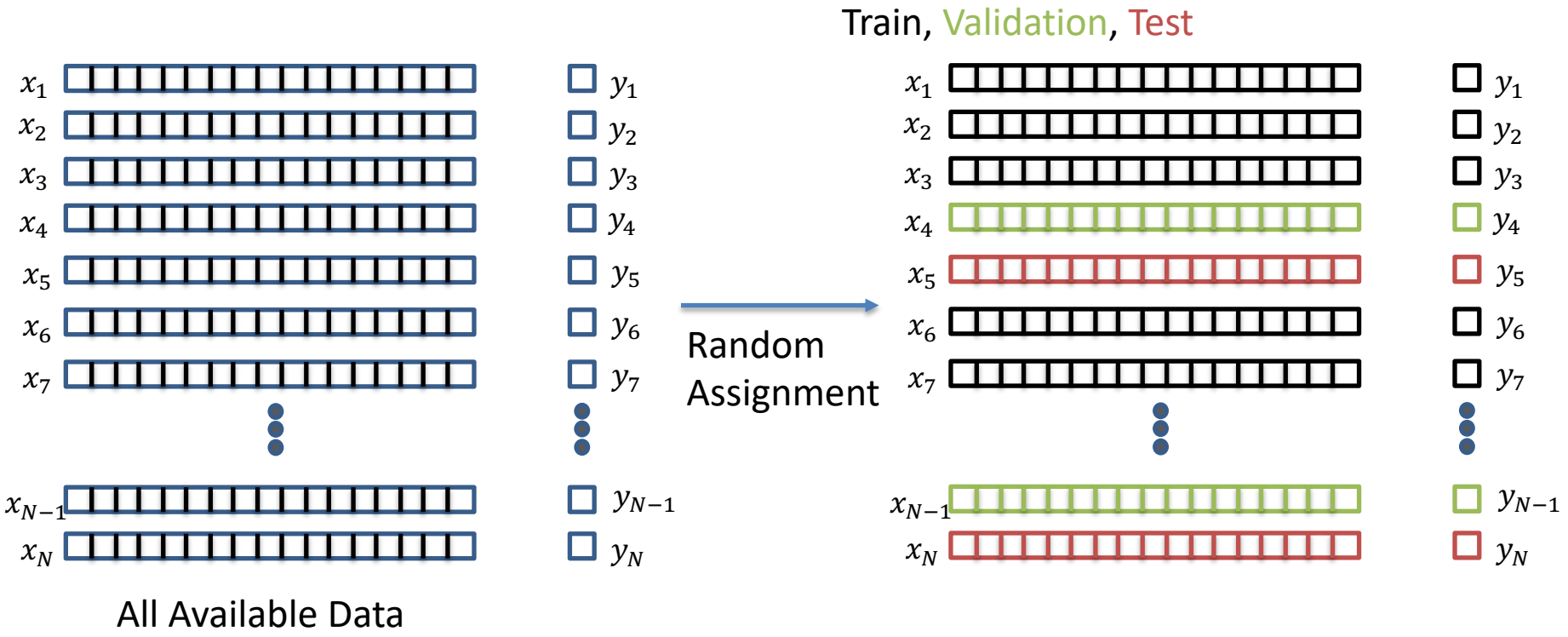
Performance in the Real World??

Standard Validation Strategy

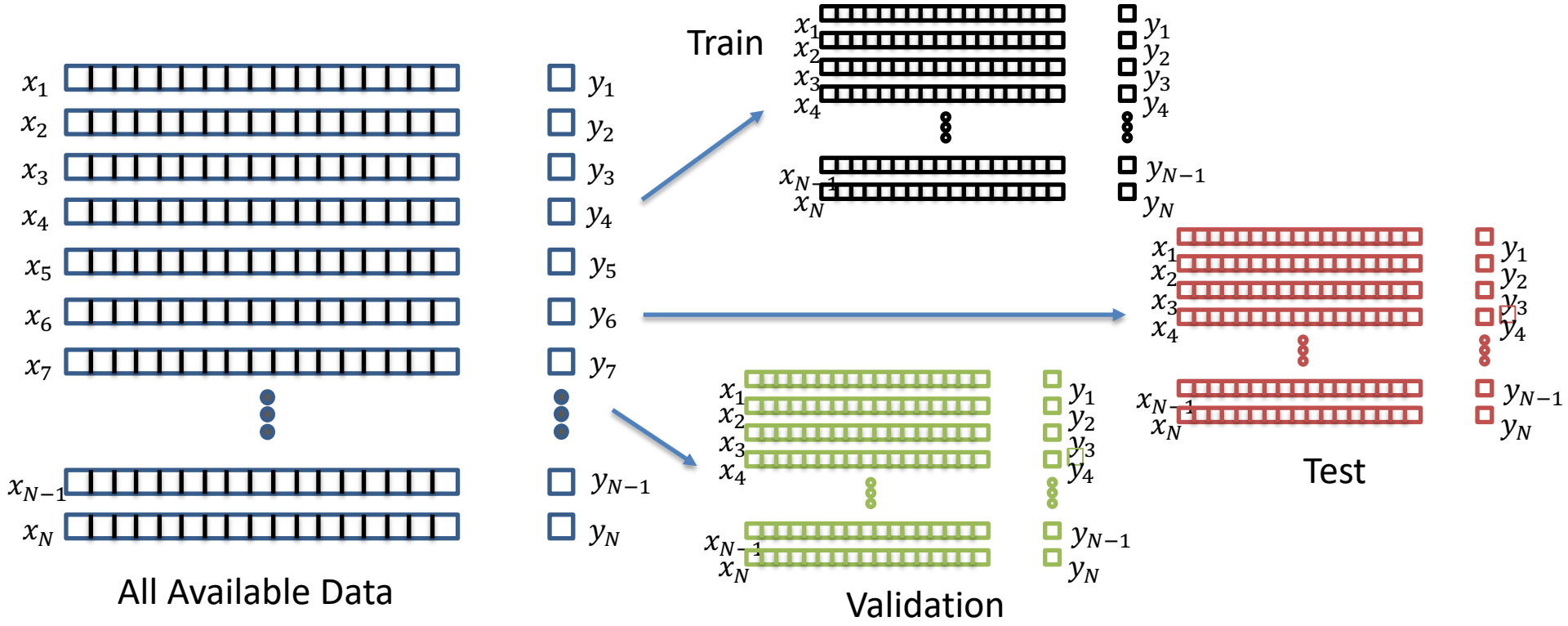
- In the end, we want to know how the network will perform *in the real world*
- Standard approach: actually try it in the real world
- This is costly; instead, can we use existing data to estimate performance?



Split Data into Separate Groups



Split Data into Separate Groups



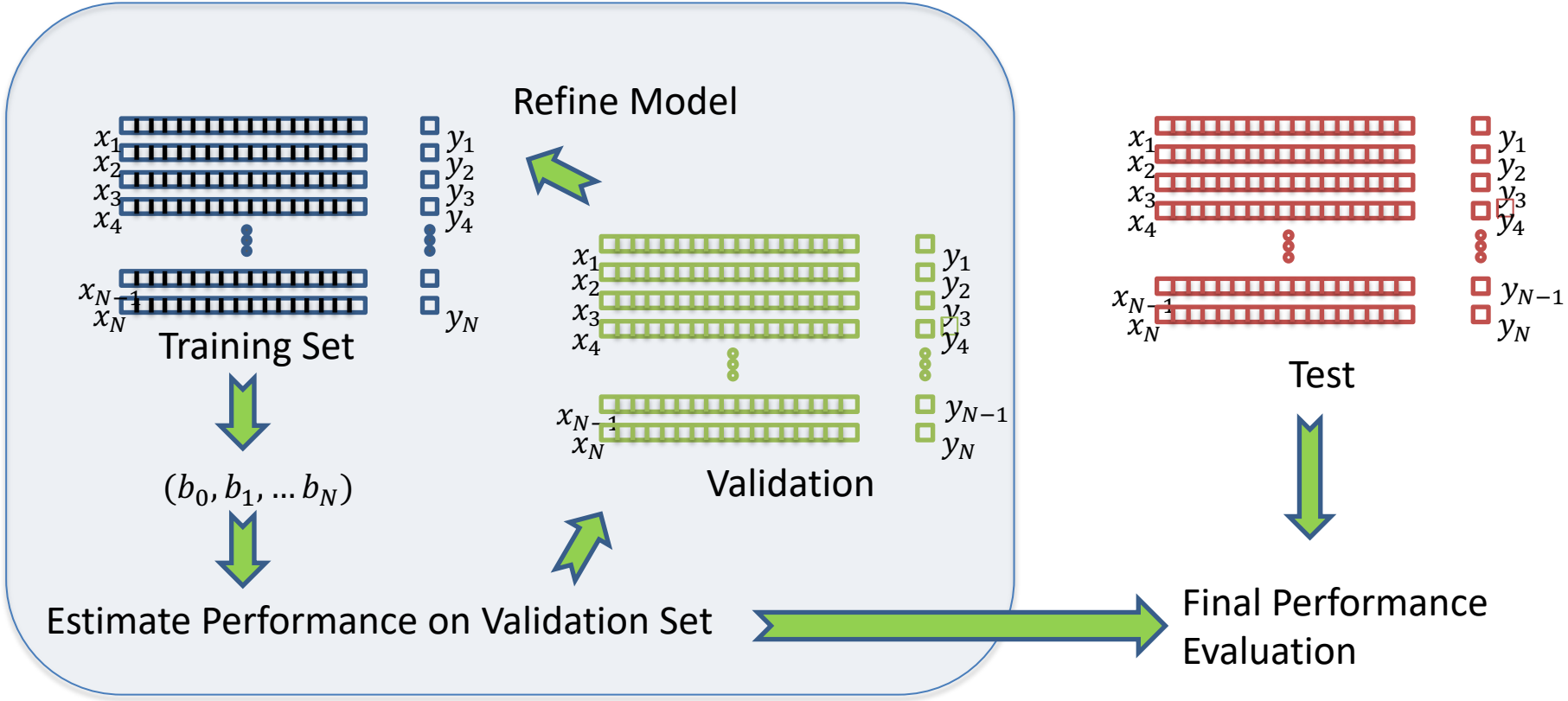
Test set

- Standard practice to create prior to any analysis—**will not be used to learn or fit any parameters**
- After learning the network, can evaluate its performance on the test set
  - This data was not included in the training/fitting, so it is analogous to running a new synthetic experiment

- Ideally, the test set will be used *once*.
  - Reusing the test set leads to bias; performance estimates will be optimistic

Validation Set

- Want to be able to compare which approach is best
  - Problematic if we only want to use a test set once
  - Can create a second held-out dataset
- The validation data is not used for learning parameters, but can be used repeatedly to estimate performance of a model
- We can pick the model with the best performance on the validation set, and run a final evaluation on the test data

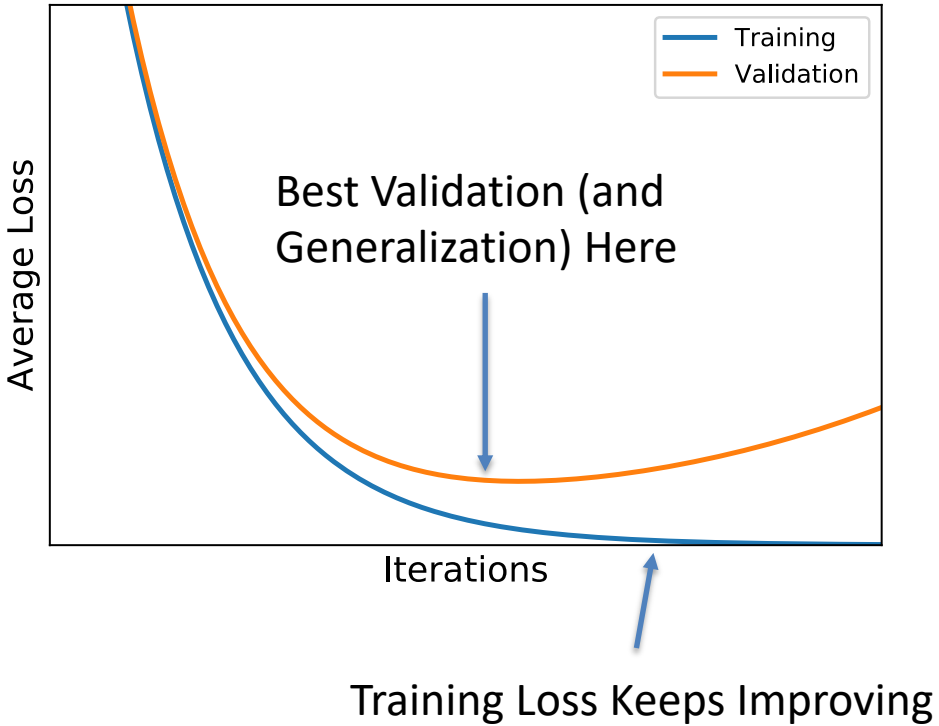

Refine Model
x_1 x_2 x_3 x_4 x_{N-1} x_N
y_1 y_2 y_3 y_4 y_N
Training Set
$(b_0, b_1, \ldots b_N)$
x_1 x_2 x_3 x_4 x_{N-1} x_N
y_1 y_2 y_3 y_4 y_{N-1} y_N
Validation
Estimate Performance on Validation Set
x_1 x_2 x_3 x_4 x_{N-1} x_N
y_1 y_2 y_3 y_4 y_{N-1} y_N
Test
Final Performance Evaluation

Validation During Optimization

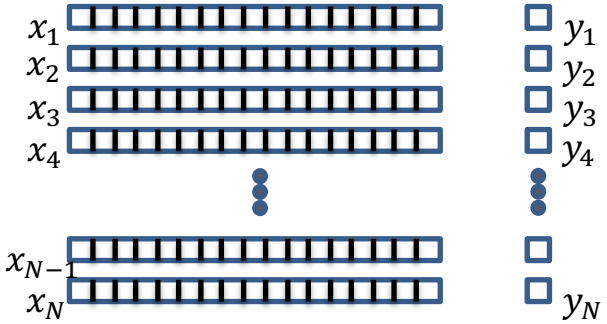
- We want to maximize the *generalization* of the network, not our previous optimization goal
- In practice, can we validate the model while running the optimization loop?

Early Stopping

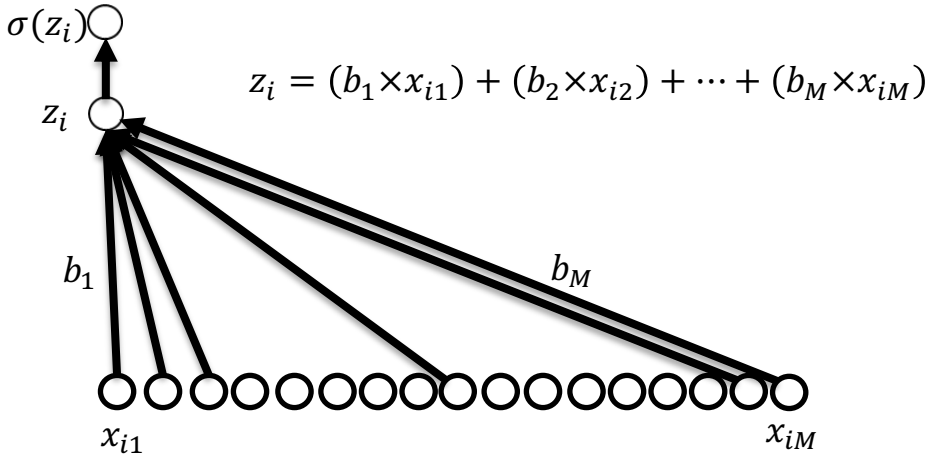
- During optimization, we can check the validation loss as we go.
- Instead of optimizing to convergence, we can optimize until the *validation* loss stops improving
  - Saves computational cost
  - Performs better on validation (and test) sets
- Widely used technique in the field



Learning Model Parameters (Recap)



Training Set



Logistic Regression Model (or "Network")

Stochastic Gradient Descent

Learned Parameters $(b_0, b_1, \ldots b_N)$

Conclusions/Next Steps

- Stochastic Gradient methods are the *modus operandi* of learning deep networks
- Proper model validation is critical to estimate real-world performance
- Hands on section you will implement stochastic gradient descent and a multi-layer perceptron

Thanks for your attention!

WRAPPING UP