Practical Introduction to Neural Network Potentials Day 2:

Neural networks

Exercise #1 Review: Modeling energies with linear regression

$$\sum_{i}^{N_{elem}} w_i \cdot N_i + b$$

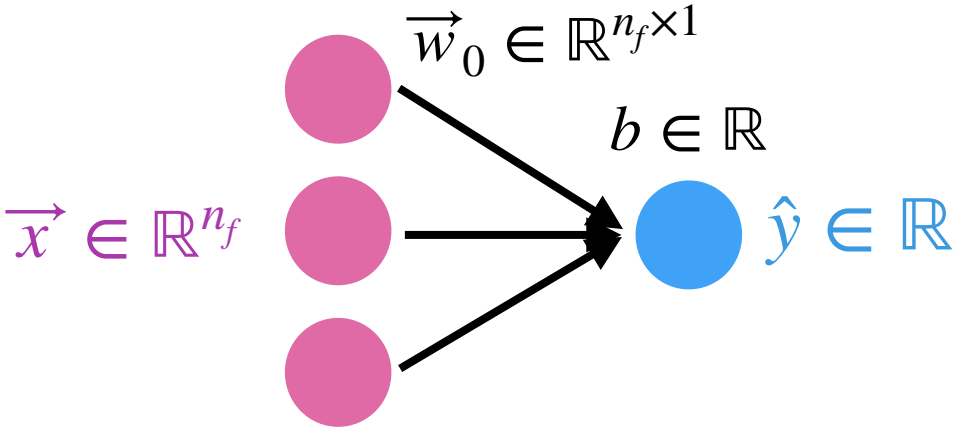
	N_H	N_C	N_O	N_{Cl}	
	2	0	1	0	$\hat{E} = 2w_H + 1w_O + b$
	3	1	0	1	$\hat{E} = 3w_H + 1w_C + 1w_{Cl} + b$

MAE ≈ 19 kcal mol^{-1}
Transferable! ~Uniform errors over 130K systems

What was the main weakness of our linear regression model?

What are neural networks?

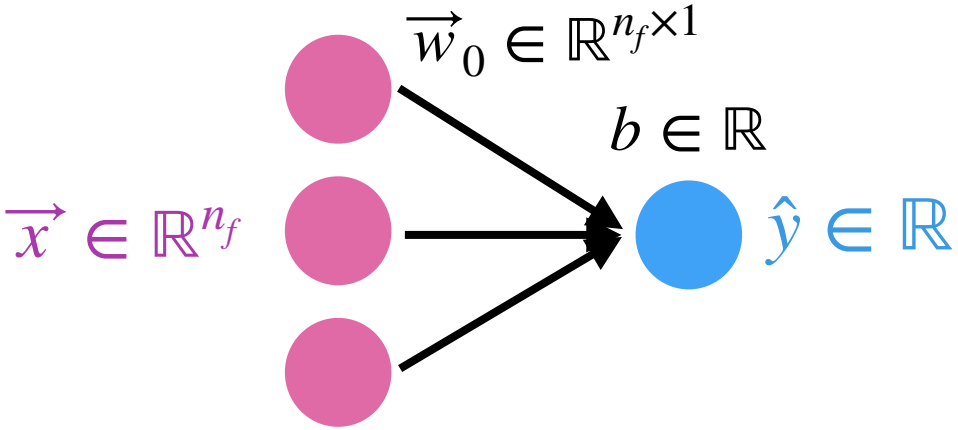
$$\hat{y} = \overrightarrow{w}_0 \cdot \overrightarrow{x} + b$$: Linear regression



What are neural networks?

$$\hat{y} = \vec{w}_0 \cdot \vec{x} + b$$: Linear regression

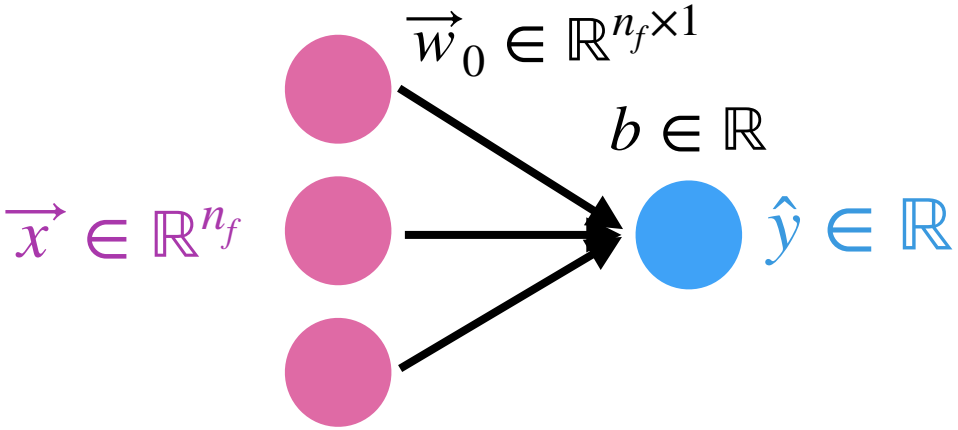
In other words, $\hat{y}$ **must be a linear combination of features** $\vec{x}$



What are neural networks?

$$\hat{y} = \sigma(\vec{w}_0 \cdot \vec{x} + b)$$: “Perceptron” (nonlinear regression)

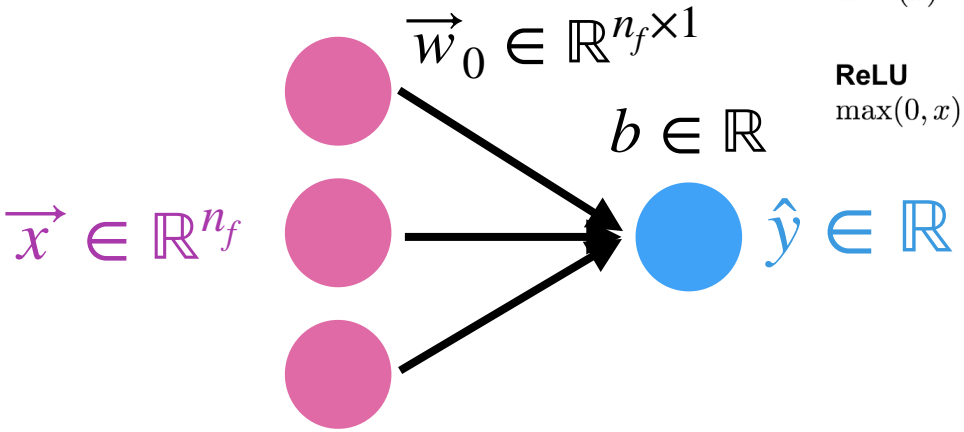
Nonlinear “activation” function



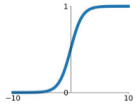
What are neural networks?

$$\hat{y} = \sigma(\vec{w}_0 \cdot \vec{x} + b)$$: “Perceptron” (nonlinear regression)

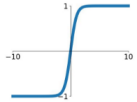
Nonlinear “activation” function



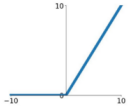
Sigmoid
$\sigma(x) = \frac{1}{1+e^{-x}}$



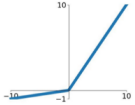
tanh
$\tanh(x)$



ReLU
$\max(0, x)$

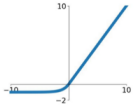


Leaky ReLU
$\max(0.1x, x)$



Maxout
$\max(w_1^T x + b_1, w_2^T x + b_2)$

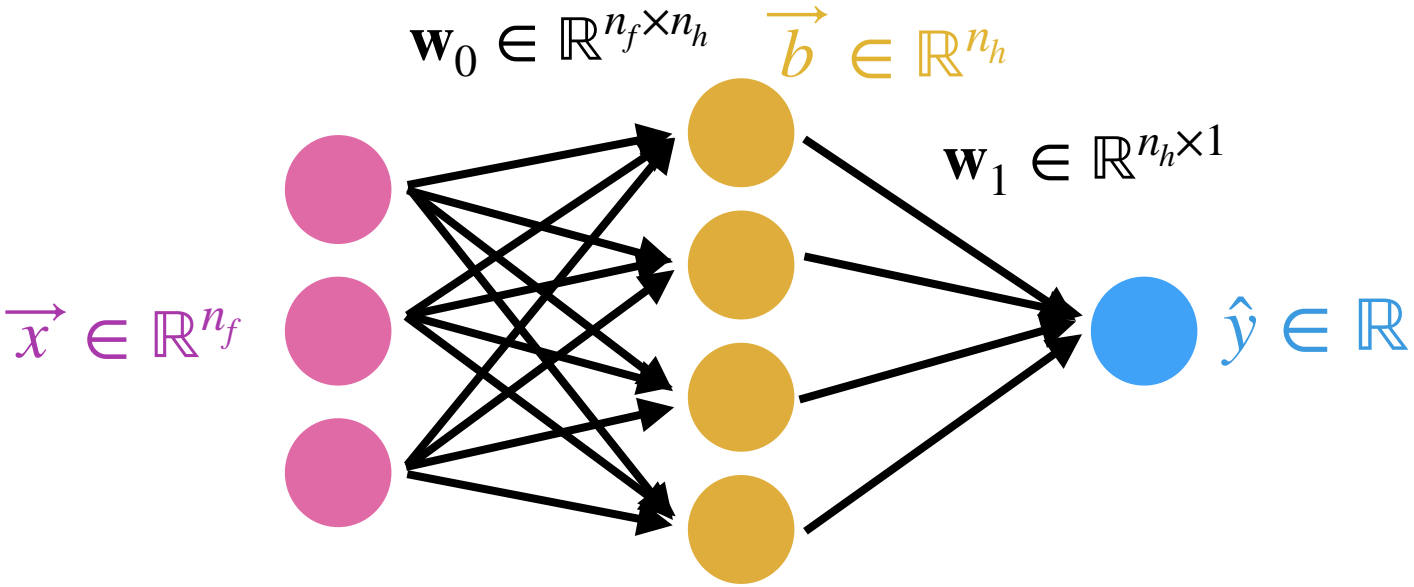
ELU
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



What are neural networks?

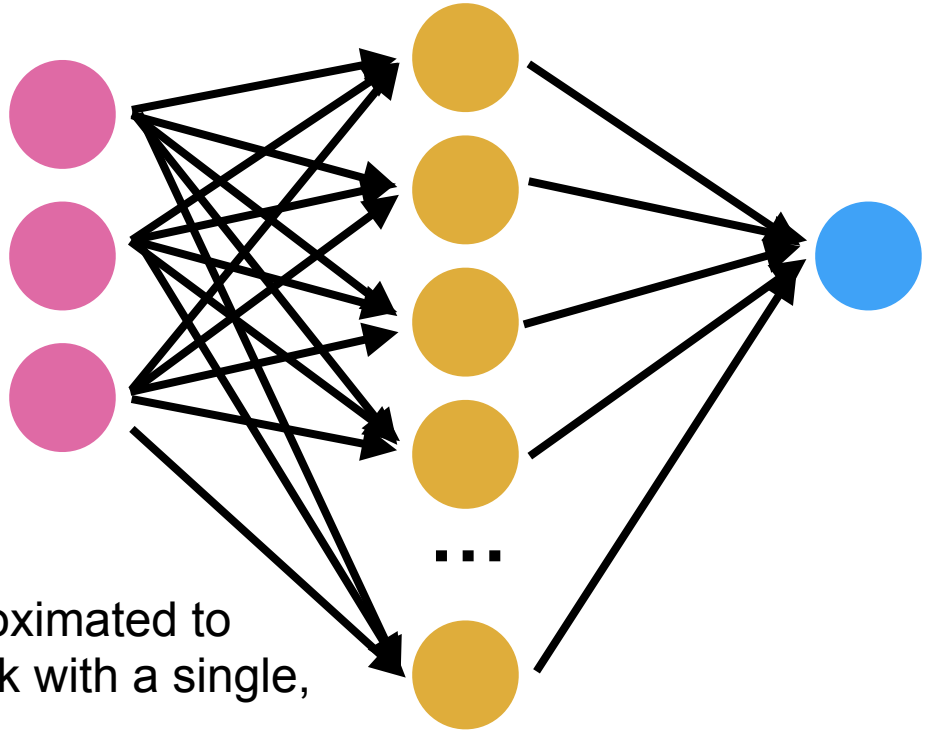
$$\hat{y} = \mathbf{w}_1 \sigma(\mathbf{w_0} \overrightarrow{x} + \overrightarrow{b})$$

: Feed-forward neural network with one hidden layer



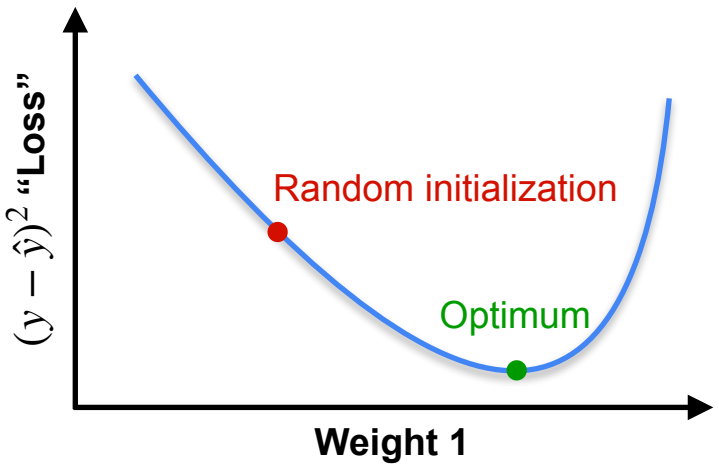
What are neural networks?

Universal approximation theorem

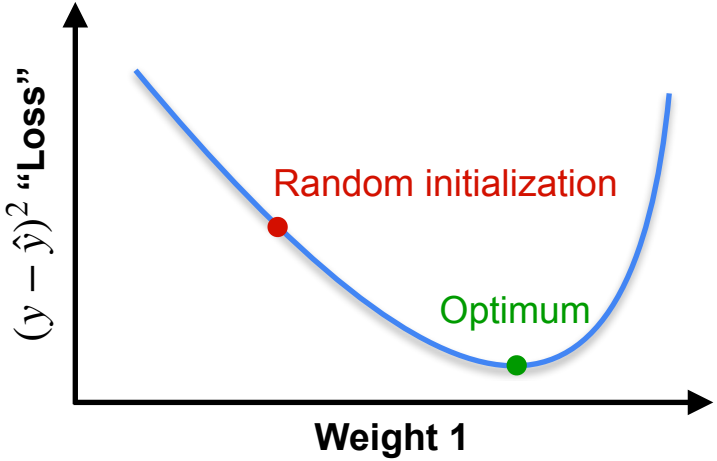


Any continuous function can be approximated to arbitrary accuracy by a neural network with a single, sufficiently large hidden layer.

Finding the parameters: The loss surface and backpropagation

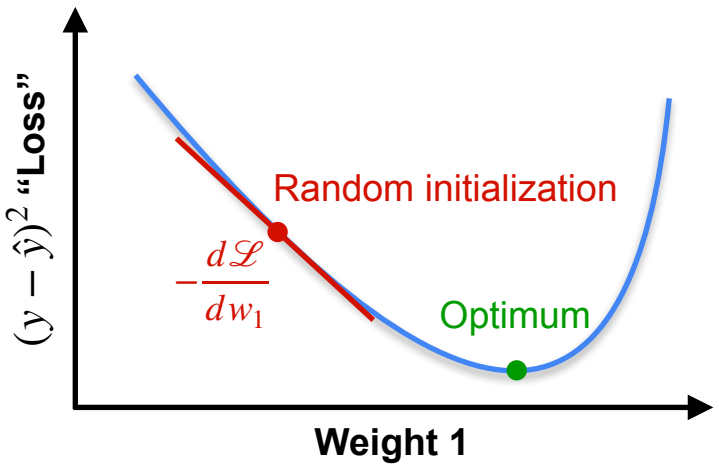


Finding the parameters: The loss surface and backpropagation



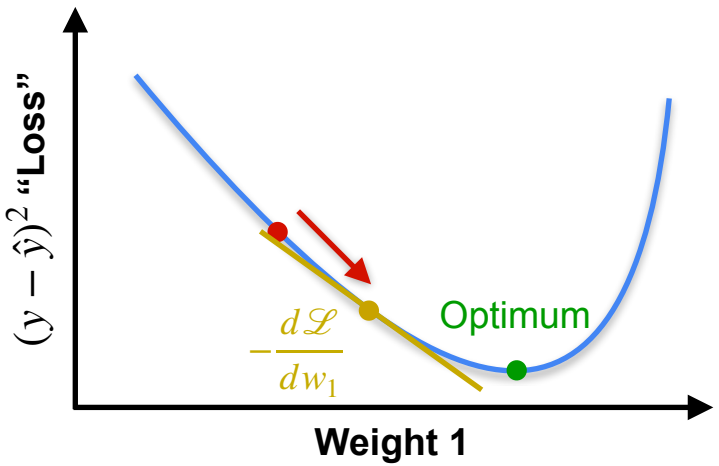
How do we set Weight 1 to the optimal value?

Finding the parameters: The loss surface and backpropagation



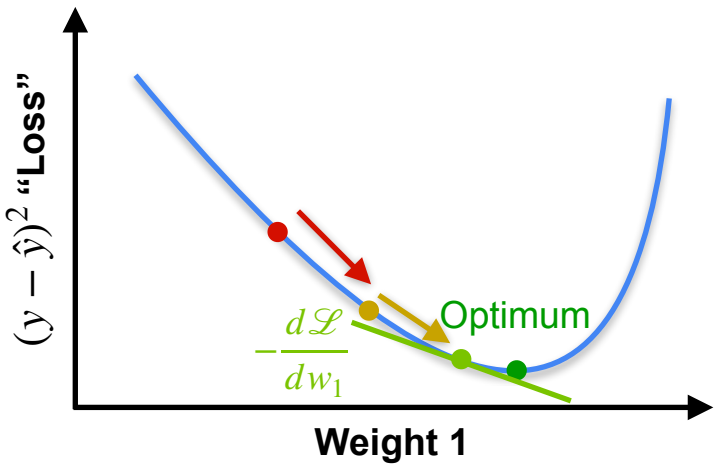
$$\mathscr{L} = (y - \hat{y})^2$$

Finding the parameters: The loss surface and backpropagation



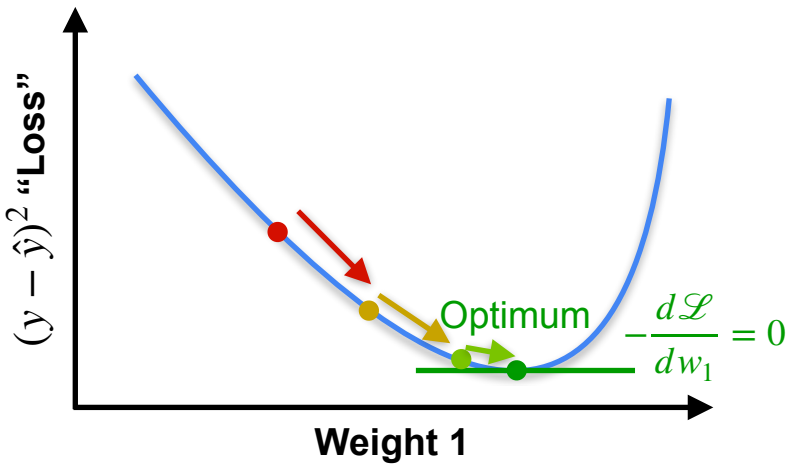
$$\mathscr{L} = (y - \hat{y})^2$$

Finding the parameters: The loss surface and backpropagation



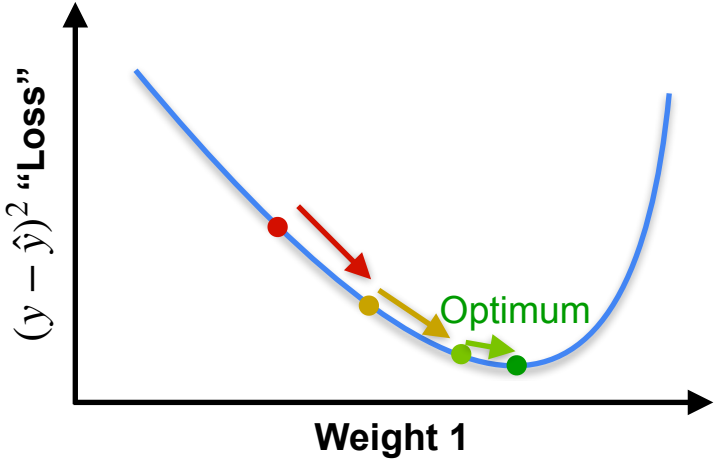
$$\mathscr{L} = (y - \hat{y})^2$$

Finding the parameters: The loss surface and backpropagation



$$\mathscr{L} = (y - \hat{y})^2$$

Finding the parameters: The loss surface and backpropagation



Reverse-mode differentiation (backpropagation) allows efficient computation of gradients of the loss function with respect to weights

Training blueprint

1. Randomly initialize weights

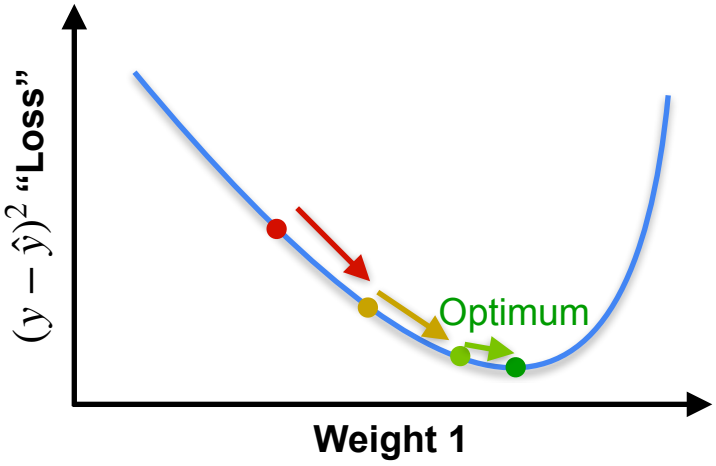
2. Feed all input data $\{\vec{x}_i\}$ to get estimates $\{\hat{y}_i\}$

3. Compute mean error $\mathscr{L} = \frac{1}{N}\sum_i^N (y_i - \hat{y}_i)^2$

4. Backpropagate to obtain derivatives $\frac{d\mathscr{L}}{d\mathbf{w}}$

5. Update weights

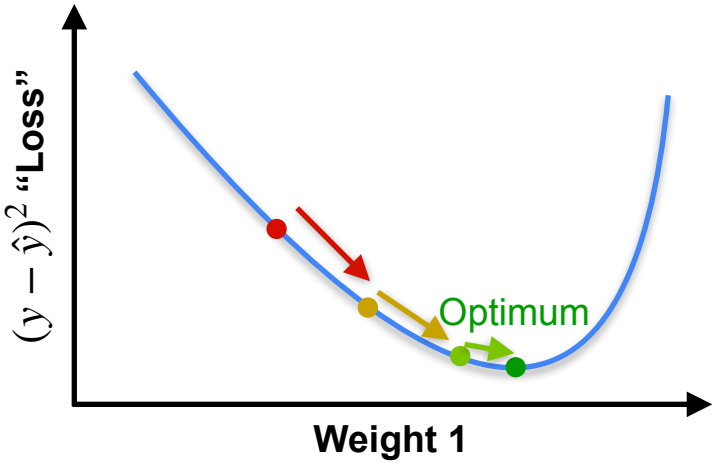
6. Repeat



Training blueprint

1. Randomly initialize weights
2. Feed all input data $\{\vec{x}_i\}$ to get estimates $\{\hat{y}_i\}$
3. Compute mean error $\mathscr{L} = \frac{1}{N}\sum_i^N (y_i - \hat{y}_i)^2$
4. Backpropagate to obtain derivatives $\frac{d\mathscr{L}}{d\mathbf{w}}$
5. Update weights
6. Repeat

1 “Epoch,” a pass through all the data

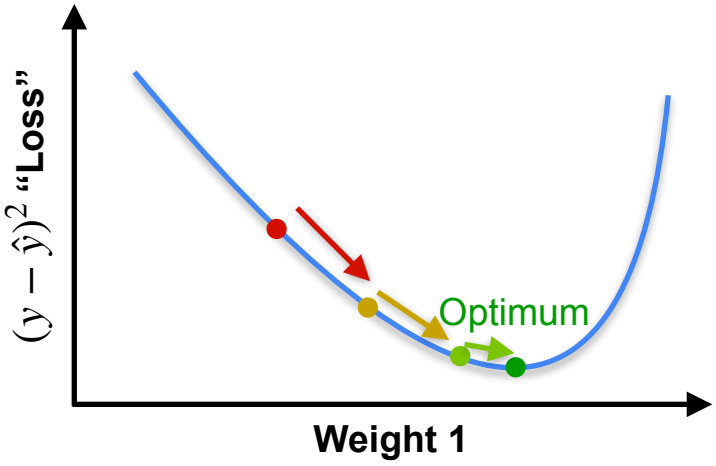


Training blueprint

1. Randomly initialize weights
2. Feed ~~all~~ a batch of input data $\{\vec{x}_i\}$ to get estimates $\{\hat{y}_i\}$
3. Compute mean error $\mathscr{L} = \frac{1}{N}\sum_i^N (y_i - \hat{y}_i)^2$
4. Backpropagate to obtain derivatives $\frac{d\mathscr{L}}{d\mathbf{w}}$
5. Update weights
6. Repeat for $\frac{N}{n}$ batches
7. Repeat for k epochs

~~1 "Epoch," a pass through all the data~~

1 "Batch," a small subset of all data n (~8-64 samples)

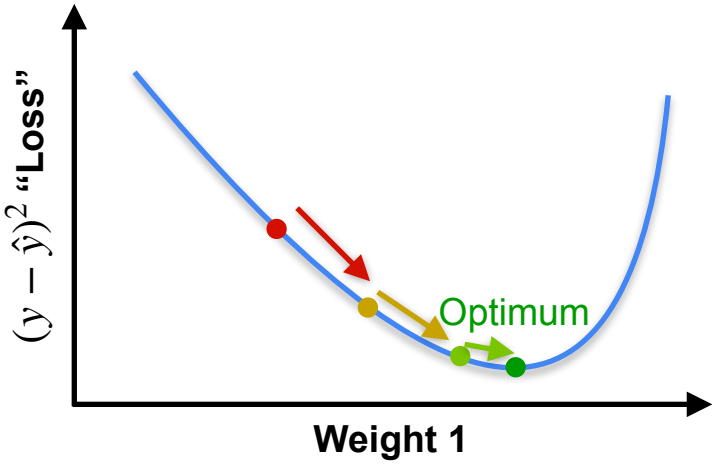


Training blueprint

1. Randomly initialize weights
2. Feed ~~all~~ a batch of input data $\{\vec{x}_i\}$ to get estimates $\{\hat{y}_i\}$
3. Compute mean error $\mathscr{L} = \frac{1}{N}\sum_i^N (y_i - \hat{y}_i)^2$
4. Backpropagate to obtain derivatives $\frac{d\mathscr{L}}{d\mathbf{w}}$
5. Update weights
6. Repeat for $\frac{N}{n}$ batches
7. Repeat for k epochs

~~1 "Epoch," a pass through all the data~~

1 "Batch," a small subset of all data n (~8-64 samples)



Batching speeds up training ($\frac{N}{n}$ gradient updates per epoch vs. 1) and requires less memory

Training blueprint (PyTorch)

```python
def train_one_epoch(epoch_index, tb_writer):
    running_loss = 0.
    last_loss = 0.

    # Here, we use enumerate(training_loader) instead of
    # iter(training_loader) so that we can track the batch
    # index and do some intra-epoch reporting
    for i, data in enumerate(training_loader):
        # Every data instance is an input + label pair
        inputs, labels = data

        # Zero your gradients for every batch!
        optimizer.zero_grad()

        # Make predictions for this batch
        outputs = model(inputs)

        # Compute the loss and its gradients
        loss = loss_fn(outputs, labels)
        loss.backward()

        # Adjust learning weights
        optimizer.step()

        # Gather data and report
        running_loss += loss.item()
        if i % 1000 == 999:
            last_loss = running_loss / 1000 # loss per batch
            print('  batch {} loss: {}'.format(i + 1, last_loss))
            tb_x = epoch_index * len(training_loader) + i + 1
            tb_writer.add_scalar('Loss/train', last_loss, tb_x)
            running_loss = 0.

    return last_loss
```

$\{\vec{x}_i\}, \{y_i\}$

https://pytorch.org/tutorials/beginner/introyt/trainingyt.html

Training blueprint (PyTorch)

```python
def train_one_epoch(epoch_index, tb_writer):
    running_loss = 0.
    last_loss = 0.

    # Here, we use enumerate(training_loader) instead of
    # iter(training_loader) so that we can track the batch
    # index and do some intra-epoch reporting
    for i, data in enumerate(training_loader):
        # Every data instance is an input + label pair
        inputs, labels = data

        # Zero your gradients for every batch!
        optimizer.zero_grad()

        # Make predictions for this batch
        outputs = model(inputs)

        # Compute the loss and its gradients
        loss = loss_fn(outputs, labels)
        loss.backward()

        # Adjust learning weights
        optimizer.step()

        # Gather data and report
        running_loss += loss.item()
        if i % 1000 == 999:
            last_loss = running_loss / 1000 # loss per batch
            print('  batch {} loss: {}'.format(i + 1, last_loss))
            tb_x = epoch_index * len(training_loader) + i + 1
            tb_writer.add_scalar('Loss/train', last_loss, tb_x)
            running_loss = 0.

    return last_loss
```

$$\frac{d\mathscr{L}}{d\mathbf{w}} = 0$$

https://pytorch.org/tutorials/beginner/introyt/trainingyt.html

Training blueprint (PyTorch)

```python
def train_one_epoch(epoch_index, tb_writer):
    running_loss = 0.
    last_loss = 0.

    # Here, we use enumerate(training_loader) instead of
    # iter(training_loader) so that we can track the batch
    # index and do some intra-epoch reporting
    for i, data in enumerate(training_loader):
        # Every data instance is an input + label pair
        inputs, labels = data

        # Zero your gradients for every batch!
        optimizer.zero_grad()

        # Make predictions for this batch
        outputs = model(inputs)

        # Compute the loss and its gradients
        loss = loss_fn(outputs, labels)
        loss.backward()

        # Adjust learning weights
        optimizer.step()

        # Gather data and report
        running_loss += loss.item()
        if i % 1000 == 999:
            last_loss = running_loss / 1000 # loss per batch
            print('  batch {} loss: {}'.format(i + 1, last_loss))
            tb_x = epoch_index * len(training_loader) + i + 1
            tb_writer.add_scalar('Loss/train', last_loss, tb_x)
            running_loss = 0.

    return last_loss
```

$$\{\hat{y}\} = \{NN(\vec{x}_i)\}$$

https://pytorch.org/tutorials/beginner/introyt/trainingyt.html

Training blueprint (PyTorch)

```python
def train_one_epoch(epoch_index, tb_writer):
    running_loss = 0.
    last_loss = 0.

    # Here, we use enumerate(training_loader) instead of
    # iter(training_loader) so that we can track the batch
    # index and do some intra-epoch reporting
    for i, data in enumerate(training_loader):
        # Every data instance is an input + label pair
        inputs, labels = data

        # Zero your gradients for every batch!
        optimizer.zero_grad()

        # Make predictions for this batch
        outputs = model(inputs)

        # Compute the loss and its gradients
        loss = loss_fn(outputs, labels)
        loss.backward()

        # Adjust learning weights
        optimizer.step()

        # Gather data and report
        running_loss += loss.item()
        if i % 1000 == 999:
            last_loss = running_loss / 1000 # loss per batch
            print('  batch {} loss: {}'.format(i + 1, last_loss))
            tb_x = epoch_index * len(training_loader) + i + 1
            tb_writer.add_scalar('Loss/train', last_loss, tb_x)
            running_loss = 0.

    return last_loss
```

$$\mathscr{L} = \frac{1}{n}\sum_{i}^{n}(y_i - \hat{y}_i)^2$$

https://pytorch.org/tutorials/beginner/introyt/trainingyt.html

Training blueprint (PyTorch)

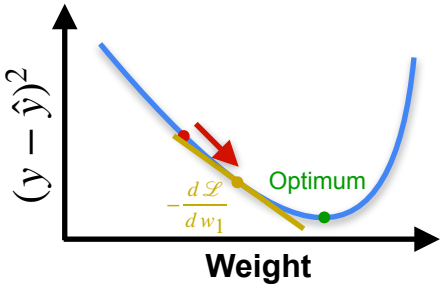
```python
def train_one_epoch(epoch_index, tb_writer):
    running_loss = 0.
    last_loss = 0.

    # Here, we use enumerate(training_loader) instead of
    # iter(training_loader) so that we can track the batch
    # index and do some intra-epoch reporting
    for i, data in enumerate(training_loader):
        # Every data instance is an input + label pair
        inputs, labels = data

        # Zero your gradients for every batch!
        optimizer.zero_grad()

        # Make predictions for this batch
        outputs = model(inputs)

        # Compute the loss and its gradients
        loss = loss_fn(outputs, labels)
        loss.backward()

        # Adjust learning weights
        optimizer.step()

        # Gather data and report
        running_loss += loss.item()
        if i % 1000 == 999:
            last_loss = running_loss / 1000 # loss per batch
            print('  batch {} loss: {}'.format(i + 1, last_loss))
            tb_x = epoch_index * len(training_loader) + i + 1
            tb_writer.add_scalar('Loss/train', last_loss, tb_x)
            running_loss = 0.

    return last_loss
```

$$\frac{d\mathcal{L}}{d\mathbf{w}}$$

https://pytorch.org/tutorials/beginner/introyt/trainingyt.html

Training blueprint (PyTorch)

```python
def train_one_epoch(epoch_index, tb_writer):
    running_loss = 0.
    last_loss = 0.

    # Here, we use enumerate(training_loader) instead of
    # iter(training_loader) so that we can track the batch
    # index and do some intra-epoch reporting
    for i, data in enumerate(training_loader):
        # Every data instance is an input + label pair
        inputs, labels = data

        # Zero your gradients for every batch!
        optimizer.zero_grad()

        # Make predictions for this batch
        outputs = model(inputs)

        # Compute the loss and its gradients
        loss = loss_fn(outputs, labels)
        loss.backward()

        # Adjust learning weights
        optimizer.step()

        # Gather data and report
        running_loss += loss.item()
        if i % 1000 == 999:
            last_loss = running_loss / 1000 # loss per batch
            print('  batch {} loss: {}'.format(i + 1, last_loss))
            tb_x = epoch_index * len(training_loader) + i + 1
            tb_writer.add_scalar('Loss/train', last_loss, tb_x)
            running_loss = 0.

    return last_loss
```



https://pytorch.org/tutorials/beginner/introyt/trainingyt.html

Overfitting

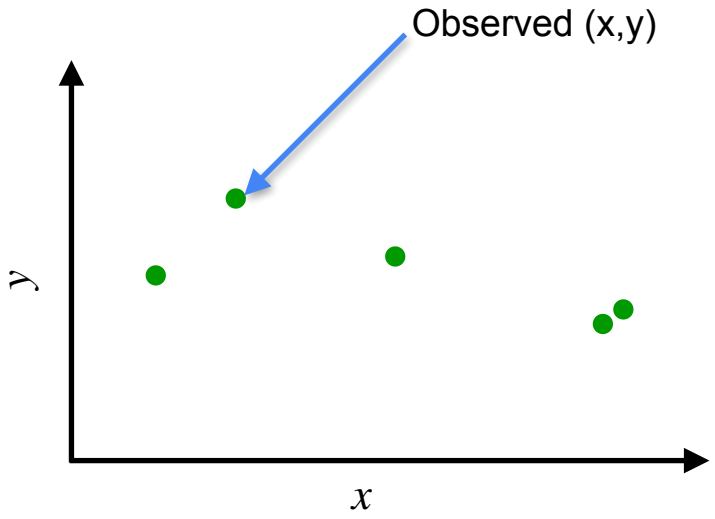
"With four parameters I can fit an elephant,
and with five I can make him wiggle his trunk."
- John von Neumann, prolific mathematician & physicist

Overfitting

"With four parameters I can fit an elephant, and with five I can make him wiggle his trunk."
- John von Neumann, prolific mathematician & physicist

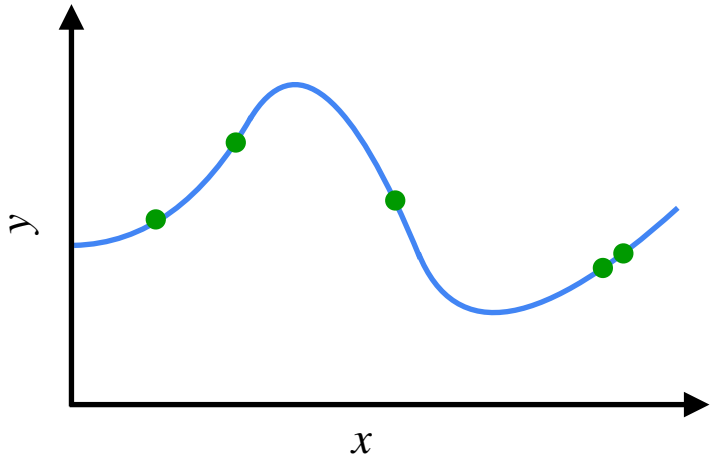
"and with fifty thousand I can approximate the solution to the Schrödinger equation on a small subset of organic molecules in vacuum"
- Derek Metcalf, equally prolific "chemist"

Overfitting

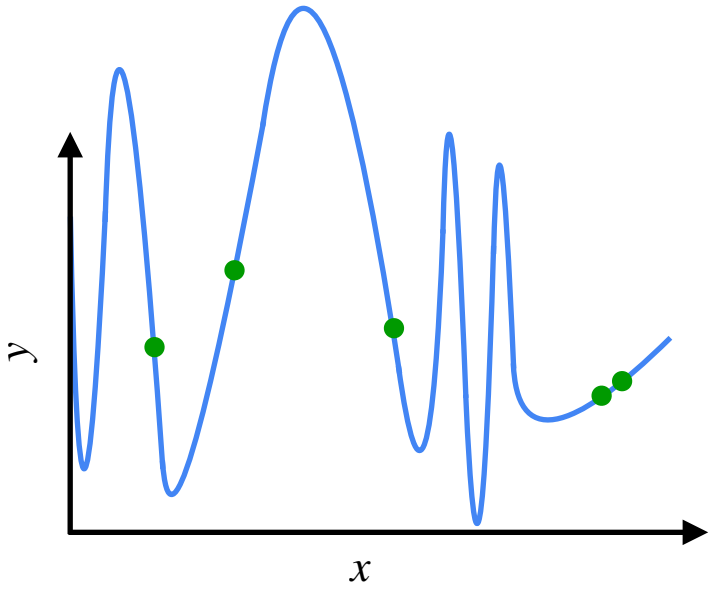


An infinite number of curves perfectly fit this data

Overfitting

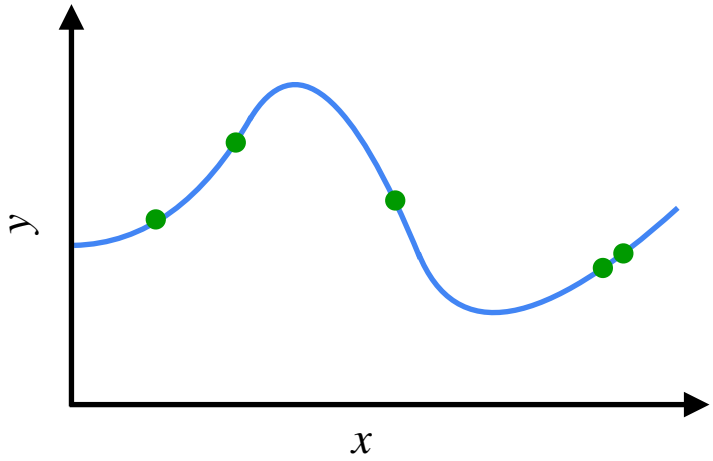


$$\mathscr{L} = 0.00$$

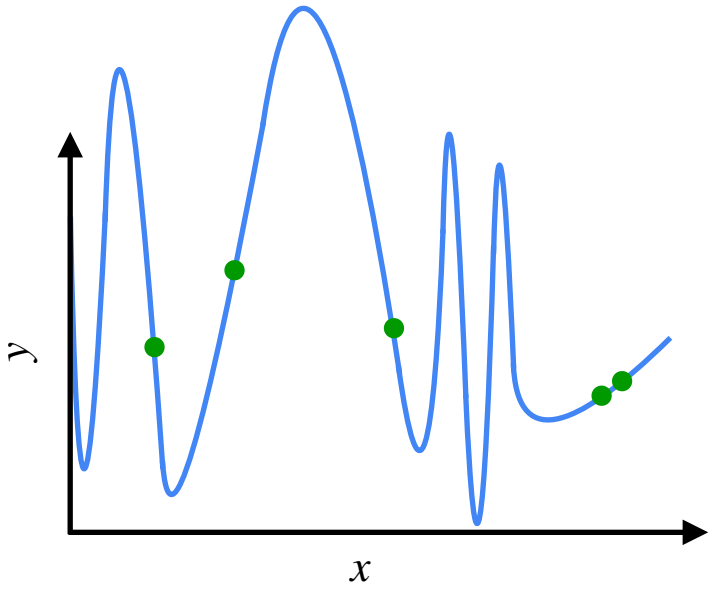


$$\mathscr{L} = 0.00$$

Overfitting



$$\mathscr{L} = 0.00$$

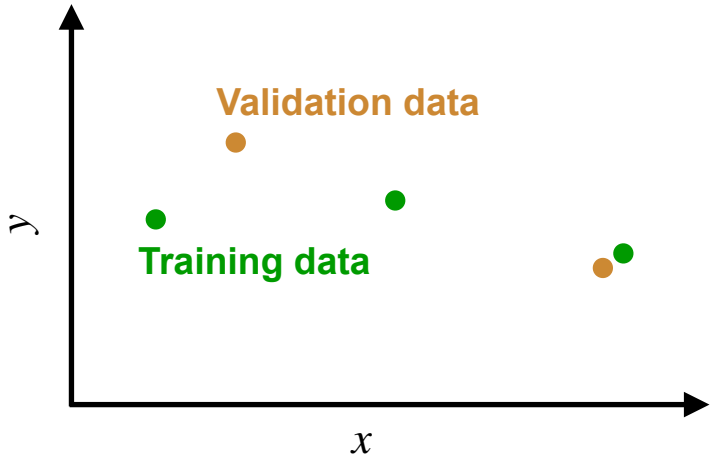


$$\mathscr{L} = 0.00$$

Which is a better model of the data?

Overfitting

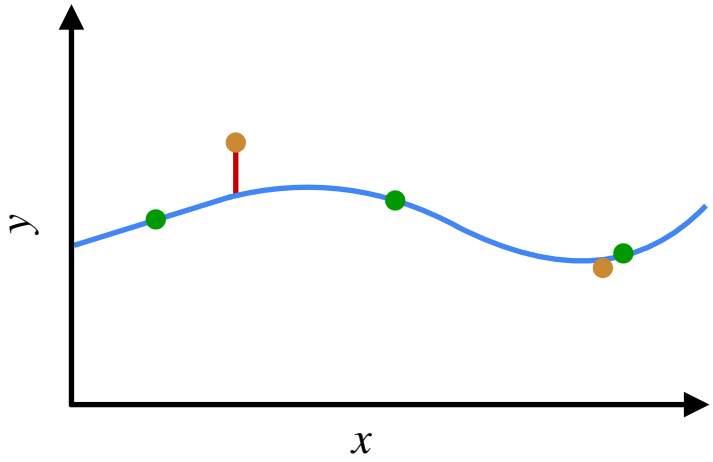
Train, Validation



Instead, we can reserve a small fraction of the data to validate the models we produce

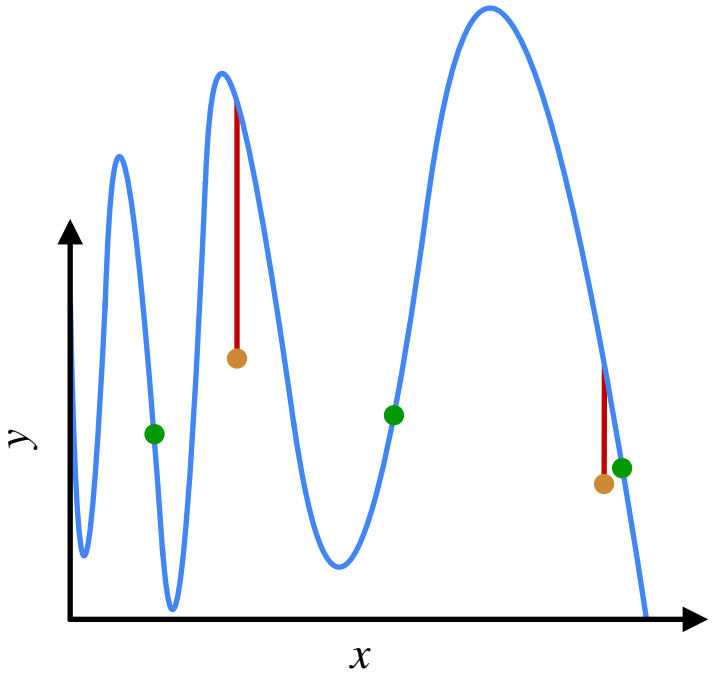
Overfitting

Train, Validation



$\mathscr{L}_t = 0.00$

$\mathscr{L}_v = 0.80$

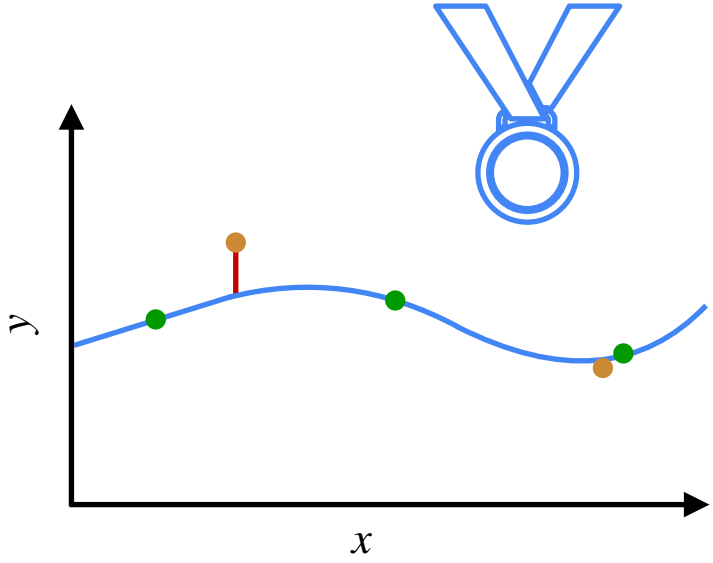


$\mathscr{L}_t = 0.00$

$\mathscr{L}_v = 4.60$

Overfitting

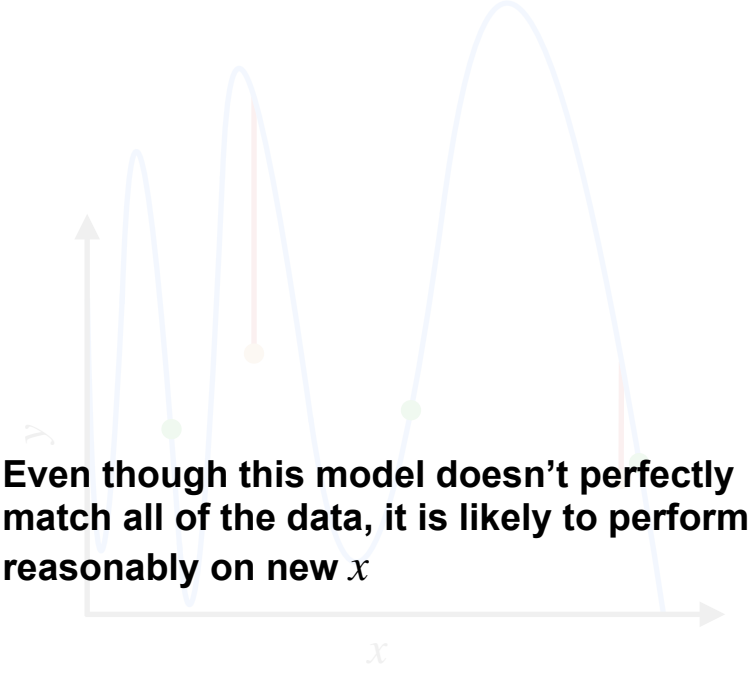
Train, Validation



$\mathscr{L}_t = 0.00$

$\mathscr{L}_v = 0.80$

Even though this model doesn't perfectly match all of the data, it is likely to perform reasonably on new x

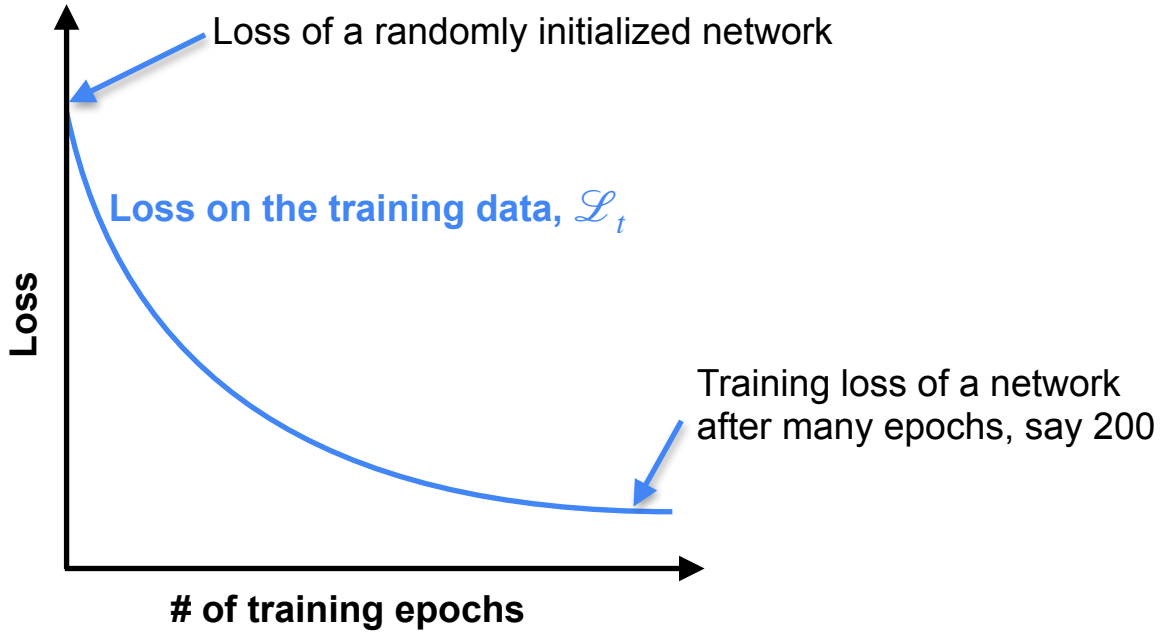


$\mathscr{L}_t = 0.00$

$\mathscr{L}_v = 4.60$

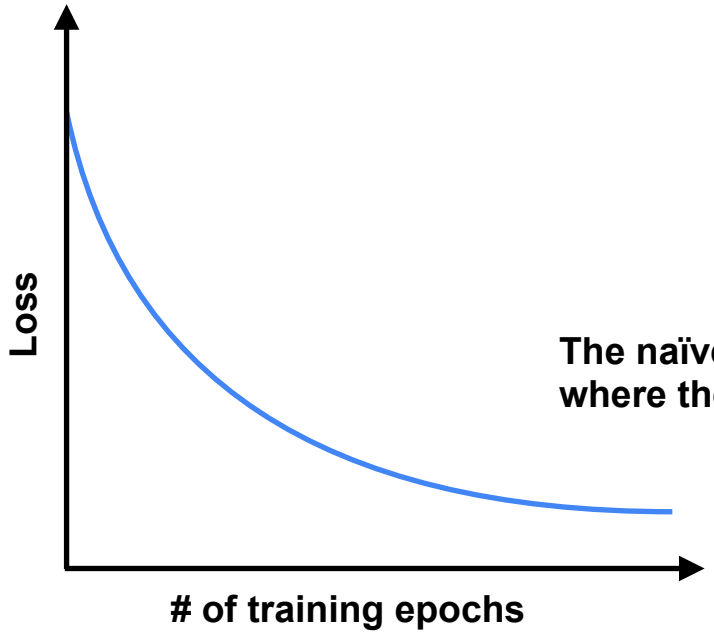
Overfitting

Detecting overfitting in practice



Overfitting

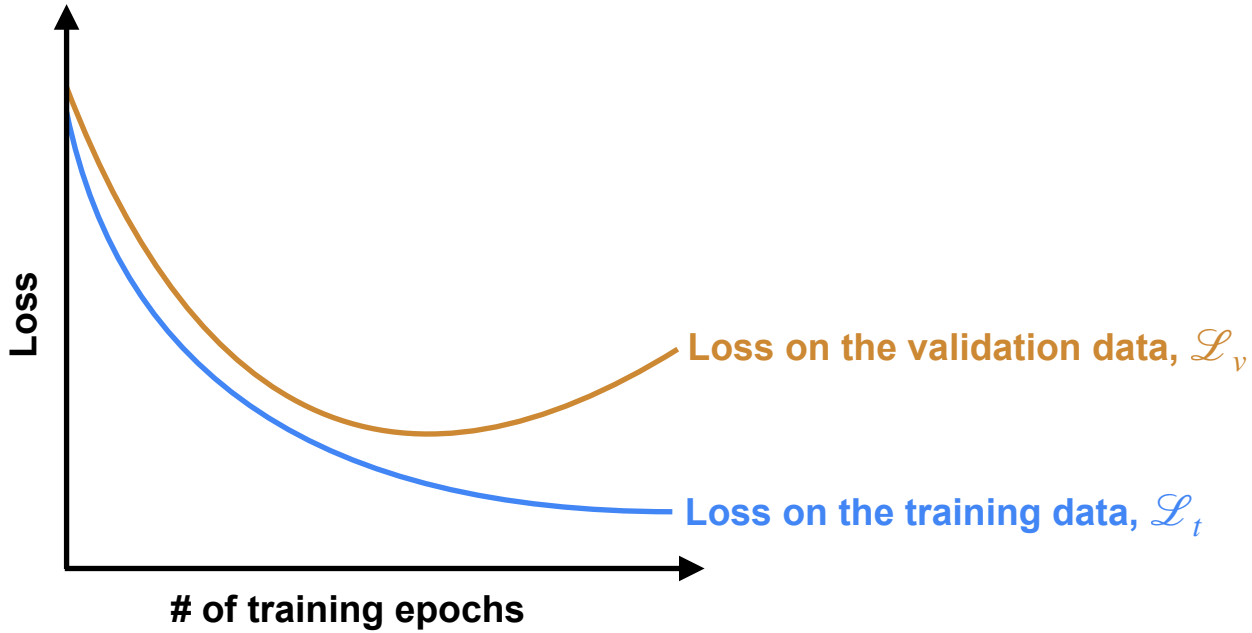
Detecting overfitting in practice



The naïve strategy is to use the model after many epochs, where the training loss has converged.

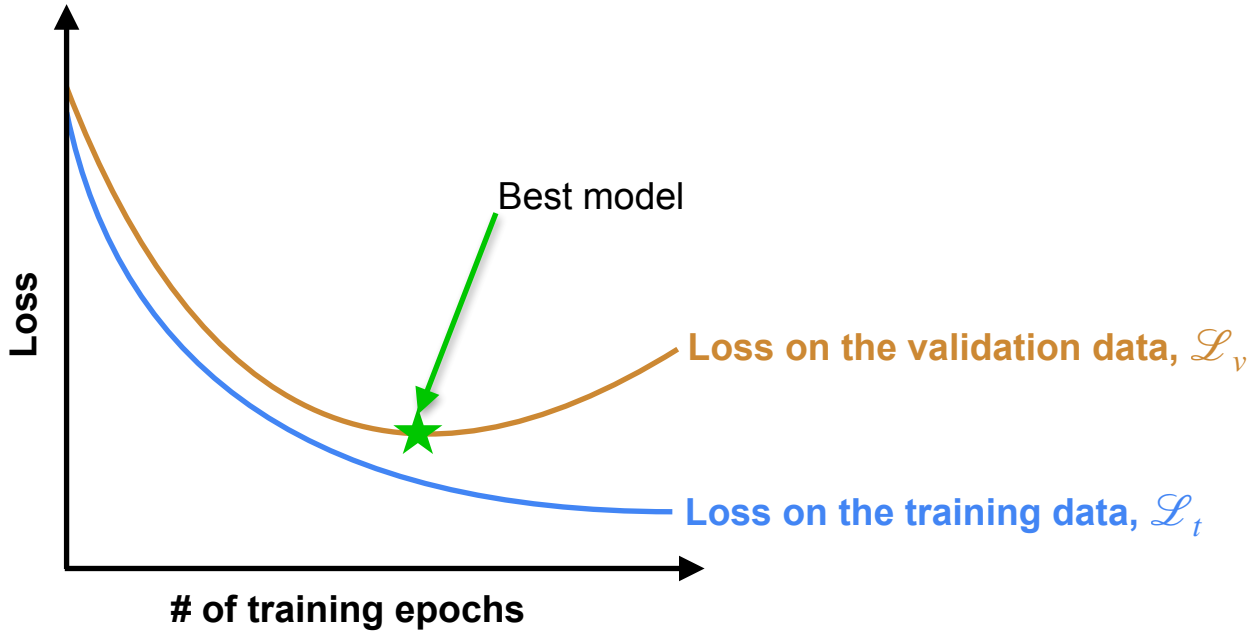
Overfitting

Detecting overfitting in practice



Overfitting

Detecting overfitting in practice



* This is a cartoon, the validation curve is not always a “U” shape. However, the general rule that you choose the model that minimizes the val. loss holds.

Hyperparameters

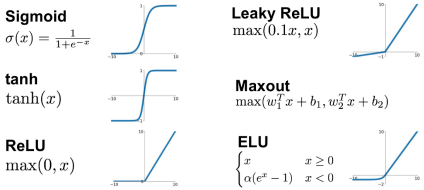
Parameters that don't change by gradient descent

Hyperparameters

Parameters that don't change by gradient descent

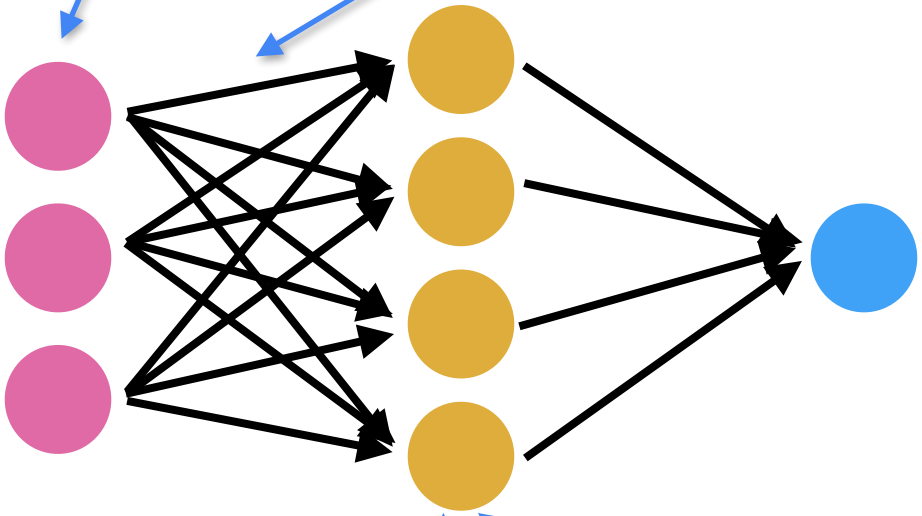
Which $\vec{x}$ to choose?

How should I initialize these? $(\mathbf{w}^{t=0})$



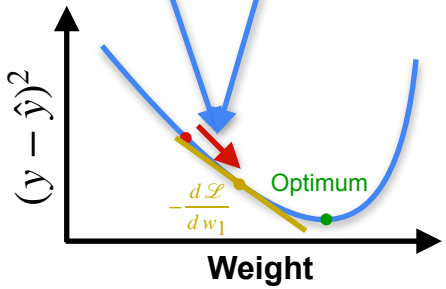
Which nonlinearity?

Does my input representation contain hyperparameters?



Which optimizer?

How big a step?



What n to choose?

How many hidden layers?

How wide? (n_h)

$$\mathscr{L} = (y - \hat{y})^2$$

Which loss function?

"Batch," a small subset of all data n (~1-64 samples, say)

Hyperparameters

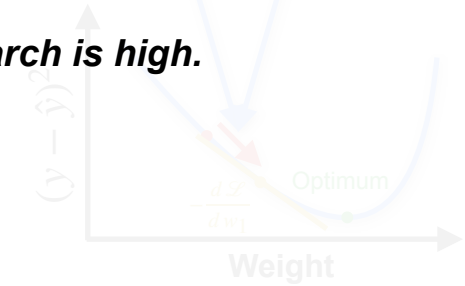
Parameters that don't change by gradient descent

Which nonlinearity?

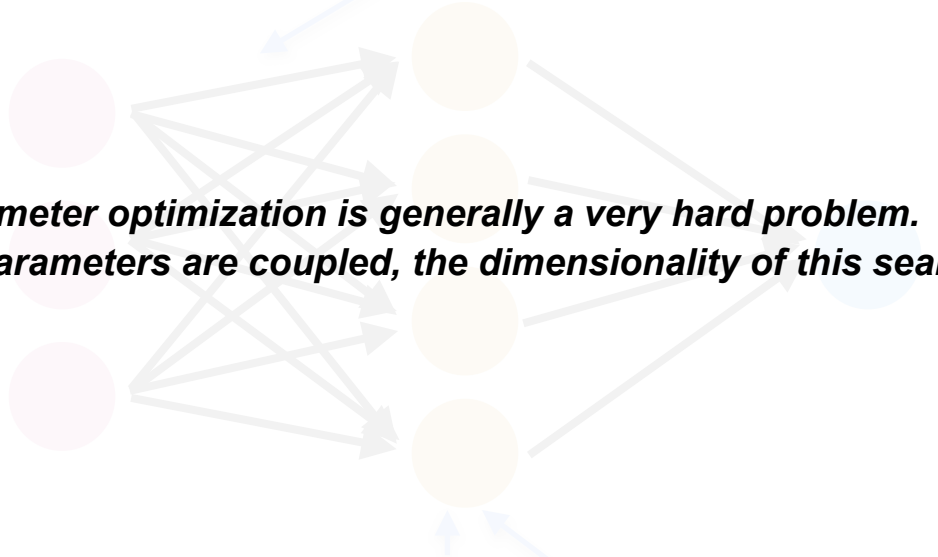
How should I initialize these? $(\mathbf{w}^{t=0})$

Which optimizer?

How big a step?



Hyperparameter optimization is generally a very hard problem.
As hyperparameters are coupled, the dimensionality of this search is high.



What n to choose?

How many hidden layers?

How wide? (n_{l_h})

"Batch," a small subset of all data n (~8-64 samples)

Hyperparameter optimization blueprint

1. For each model hyperparameter, choose a set of possible values, e.g.

Hyperparameter	Possible values
Network depth	[2, 3, 4]
Network width	[32, 64, 128]
Learning rate (LR)	[1e-5, 5e-5, 1e-4, 5e-4, 1e-3]

= 3 * 3 * 5 = 45 possible hyperparameter combinations

Hyperparameter optimization blueprint

1. For each model hyperparameter, choose a set of possible values, e.g.

Hyperparameter	Possible values
Network depth	[2, 3, 4]
Network width	[32, 64, 128]
Learning rate (LR)	[1e-5, 5e-5, 1e-4, 5e-4, 1e-3]

= 3 * 3 * 5 = 45 possible hyperparameter combinations

2. Train a neural network for each combination

3. Choose the model with the lowest validation error

Since hyperparameters don't change while training a model, we need a third data split in addition to training and validation: a "test set"

Training

~70% of data

Used to optimize network weights and biases with gradient descent

Validation

~20% of data

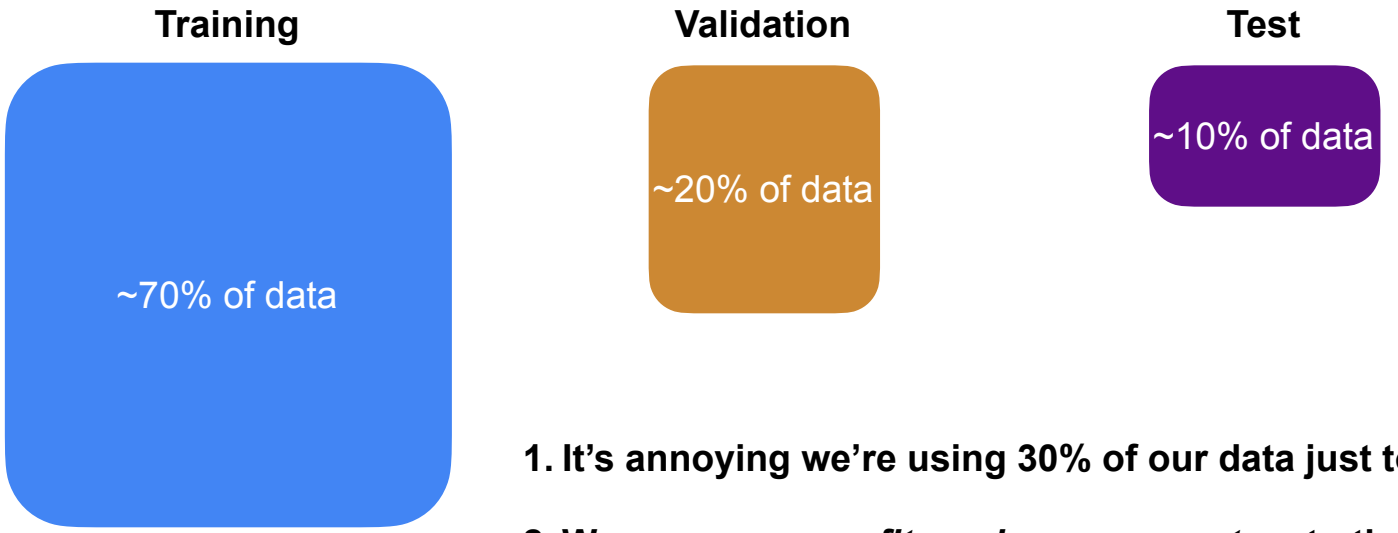
Used to choose optimal hyperparameters

Test

~10% of data

Used to evaluate *final, optimal* model quality (how we expect the model to perform in the wild)

* There is no "standard" data amount for each split, you just need to have enough val and test that model quality can be evaluated without too much noise

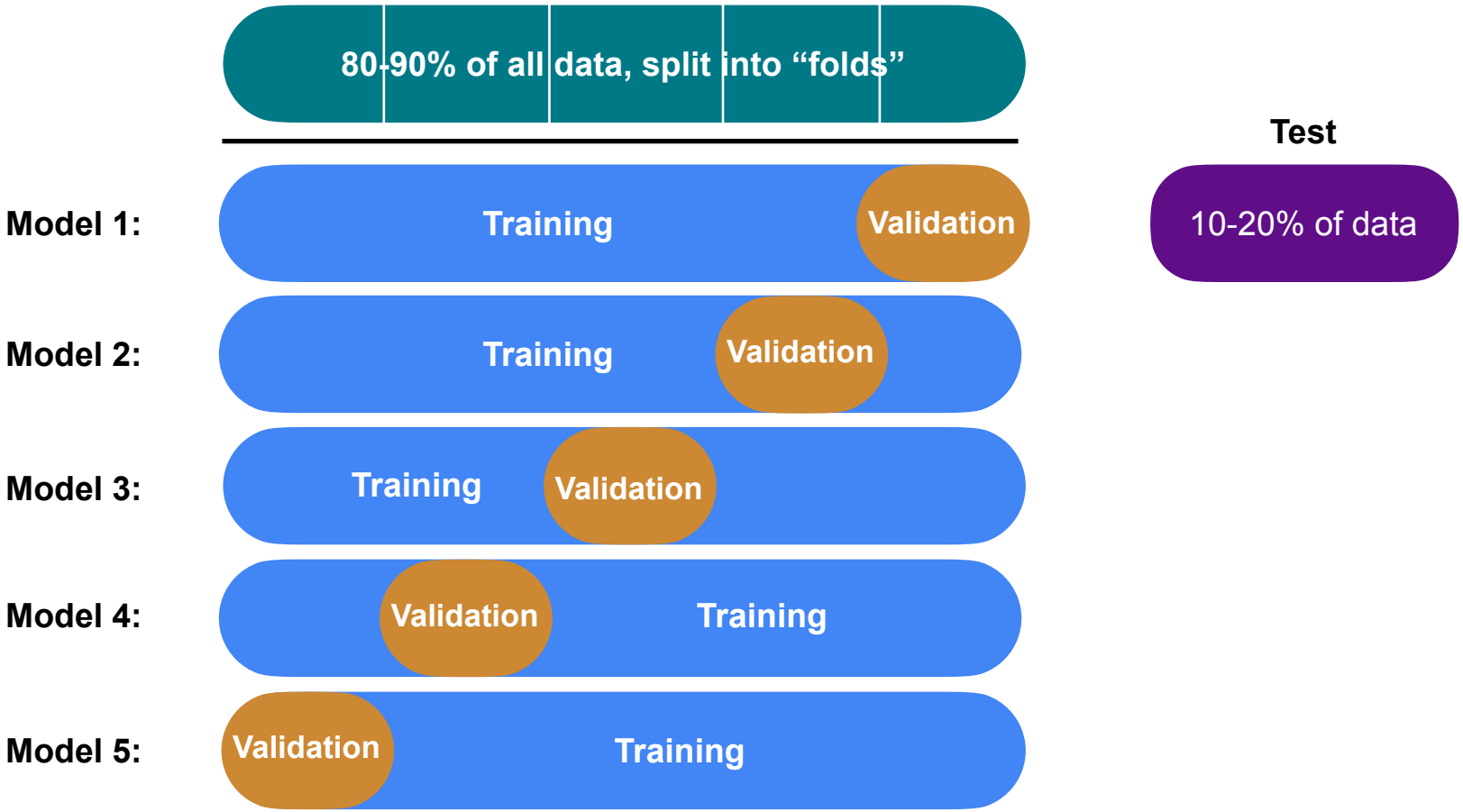


1. **It's annoying we're using 30% of our data just to validate & test**

2. **We can even *overfit our hyperparameters* to the validation set if we search hard enough**

* There is no "standard" data amount for each split, you just need to have enough val and test that model quality can be evaluated without too much noise

Cross-validation



* There is no “standard” data amount for each split, you just need to have enough val and test that model quality can be evaluated without too much noise

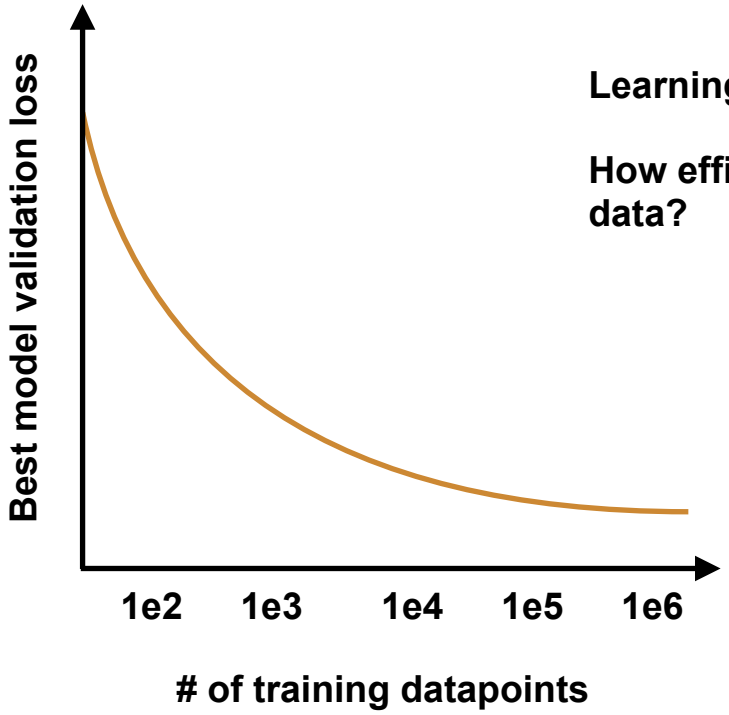
Hyperparameters

Practical, NN potential-centric advice*

Hyperparameter	Strategy
Optimizer	Use “Adam”
Network depth	Performance is usually convergent with size, start with 3 hidden layers
Network width	Performance is usually convergent with size, start with 32 nodes / layer
Learning rate (LR)	Sensitive — sample several factors of 5 or 10 around the default
Batch size	Tightly coupled to LR — choose one (8-32) and vary the LR
Activation functions	Use “SELU” (or “ReLU,” a bit cheaper but worse properties)
Loss function	Use “MSE” for regression
Variable width	Usually equal width is fine, but can play with this

* These are not gospel and are subject to change as the field evolves. There are arguments for and against every one of these recommendations on the internet. Use with caution.
* If a common hyperparameter is unlisted, I either forgot it or you should use the default.

Data efficiency & learning curves



Learning curves:

How efficient is your method at utilizing the signal in your data?

Neural network libraries*

PyTorch

scikit learn

TensorFlow

*Hot takes and conclusions based on personal experience from working in a scientific field as of July 2022, we are not responsible for your future misalignment with the field.

Neural network libraries

PyTorch

TensorFlow

- Fine-grained control over training and architecture
- Fast
- Relatively easy to write

Neural network libraries

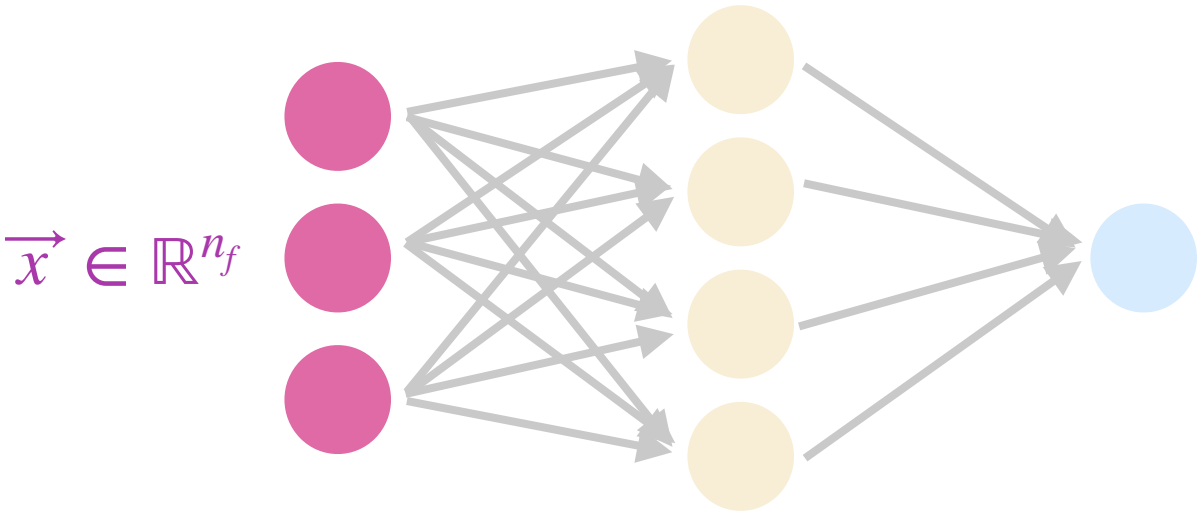
- Coarse control over neural network parameters
- Very easy to write and iterate on *simple models*
- Very hard to incorporate into complex or nonstandard workflows

Neural network libraries

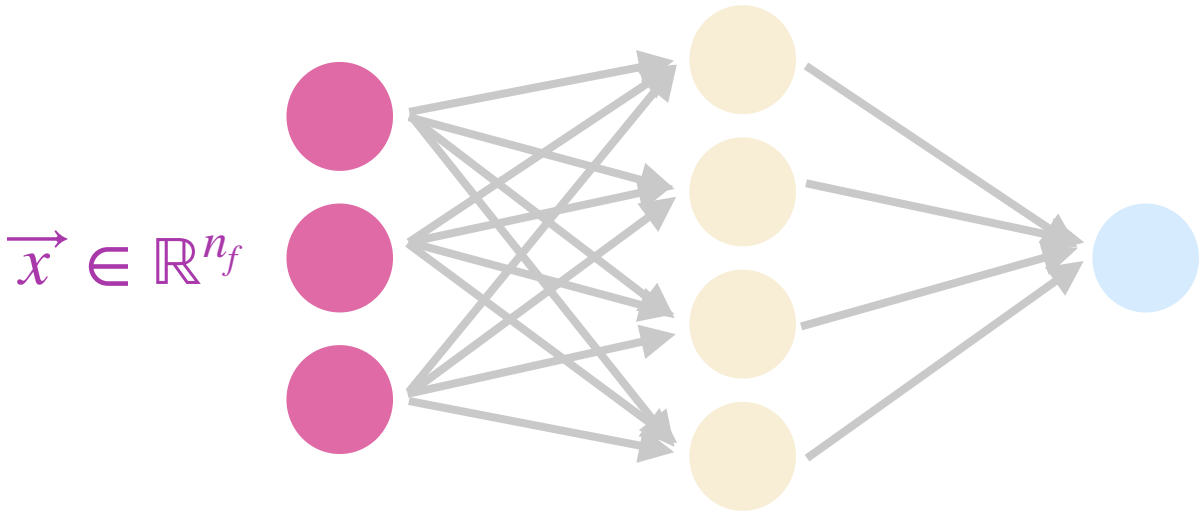
- General linear algebra library
- Best logo
- Fast & parallelizable
- Small, research-centric community

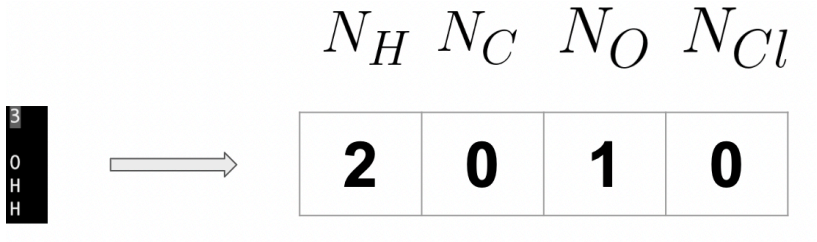
Neural network libraries

Features

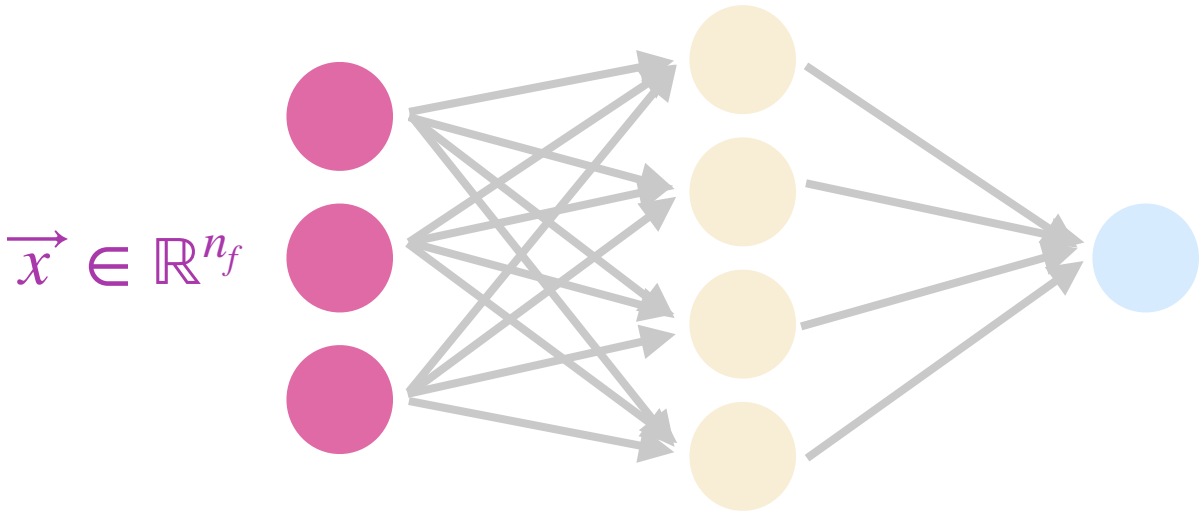


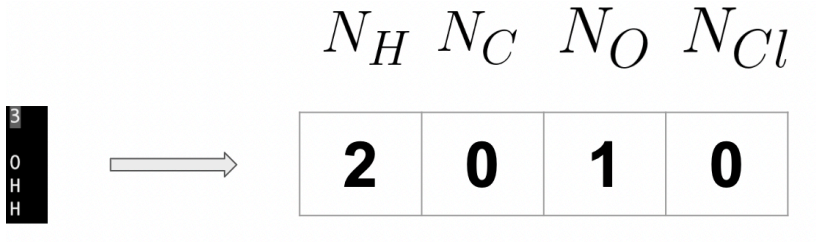
Features





Features

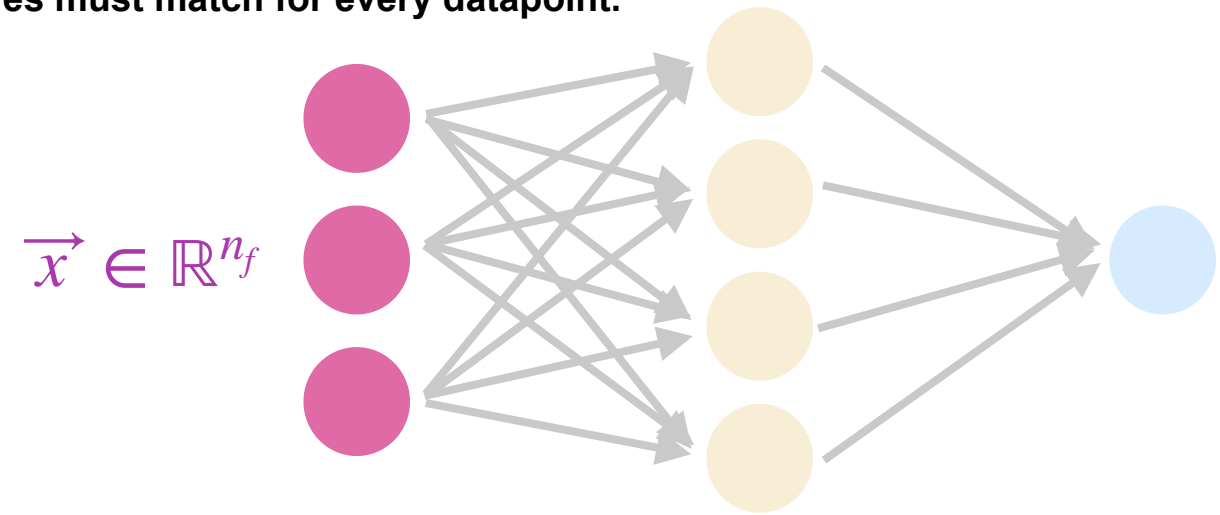


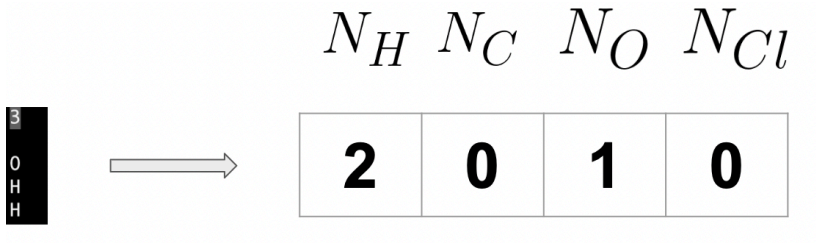


Your choice of features should reflect what you think impacts the output quantity y

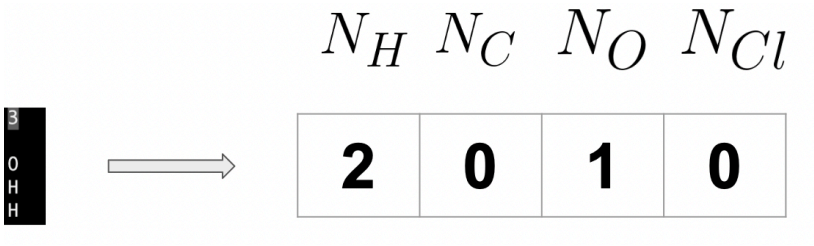
Features

Notice that, like with linear regression, the number of input features must match for every datapoint.

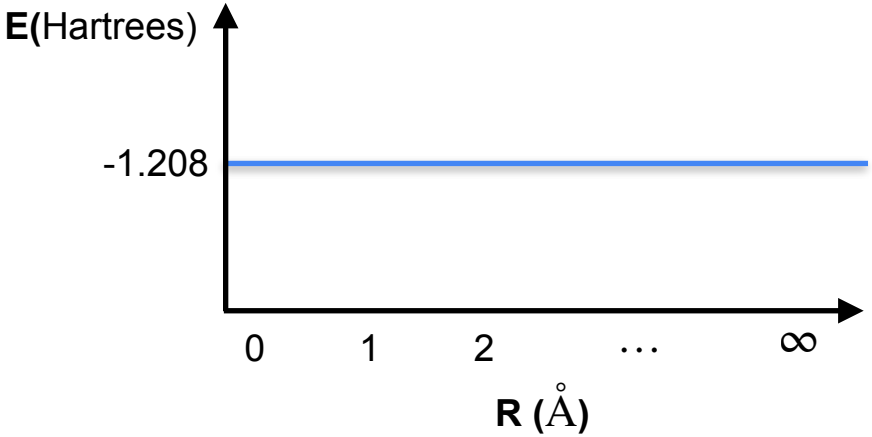




Features



Recall, this model produces the following potential energy surface (PES) for hydrogen dissociation:

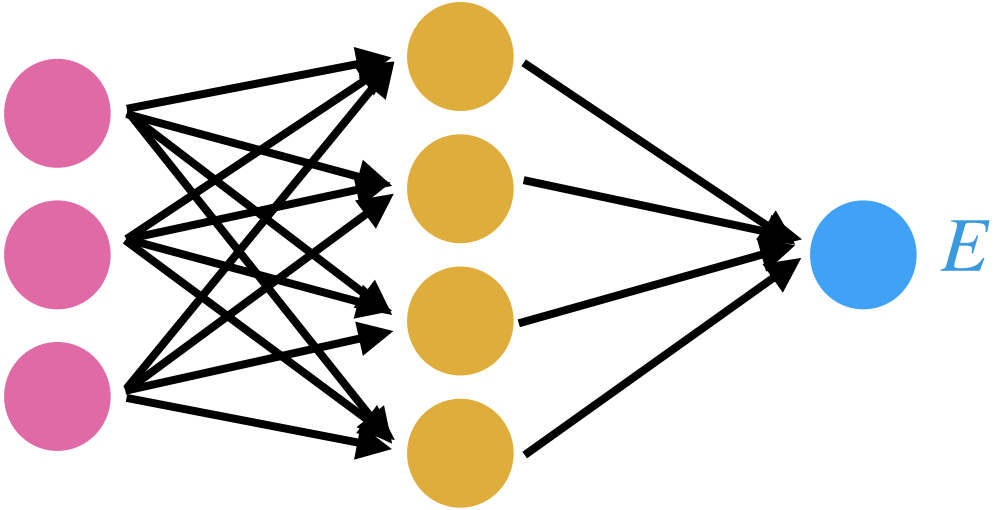


Exercise

Today, we'll bring back coordinates as we know those contain all of the info to get the energy.

```
5

C1          0.558980060028    -0.000003409116    -0.000010838753
C          -1.233013253888     0.000007518693     0.000023805641
H          -1.571271543066    -0.697791943059     0.759266356426
H          -1.571271410823     1.006427246976     0.224727628339
H          -1.571303371347    -0.308606540429    -0.983901358501
```

Raw xyz coordinates



Exercise

Today, we'll bring back coordinates as we know those contain all of the info to get the energy.

```
5

C1          0.558980060028    -0.000003409116    -0.000010838753
C          -1.233013253888     0.000007518693     0.000023805641
H          -1.571271543066    -0.697791943059     0.759266356426
H          -1.571271410823     1.006427246976     0.224727628339
H          -1.571303371347    -0.308606540429    -0.983901358501
```