CS/DS 541: Class 3

Jacob Whitehill

Hyperparameter tuning

Hyperparameter tuning

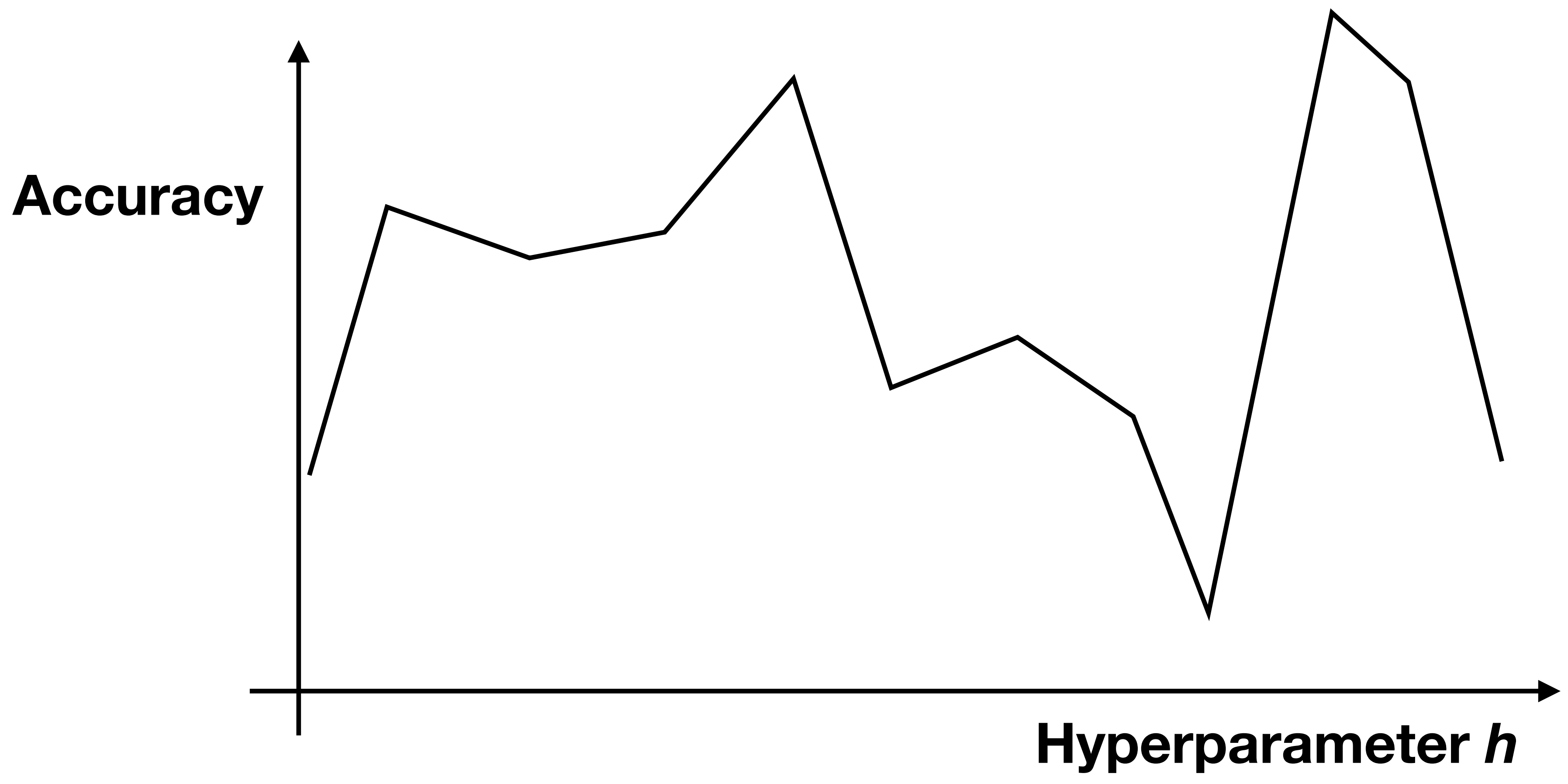
- The values we **optimize** when training a machine learning model — e.g., **w** and *b* for linear regression — are the **parameters** of the model.

- There are also values related to the training process itself — e.g., learning rate ε, batch size $\tilde{n}$, regularization strength α — which are the **hyperparameters** of training.

Hyperparameter tuning

- If you choose hyperparameters on the test set, you are likely deceiving yourself about how good your model is.

- **This is a subtle but very dangerous form of ML cheating.**



Hyperparameter tuning

- Instead, you should use a separate dataset that is not part of the test set to choose hyperparameters.
- Two commonly used (and rigorous) approaches:
    - Training/validation/testing sets
    - Double cross-validation

Training/validation/testing sets

- In an application domain with a large dataset (e.g., 100K examples), it is common to partition it into three subsets:
  - Training (typically 70-80%): optimization of parameters
  - Validation (typically 5-10%): tuning of hyperparameters
  - Testing (typically 5-10%): evaluation of the final model
- For comparison with other researchers' methods, this partition should be fixed.

Training/validation/testing sets

- Hyperparameter tuning works as follows:

  1. For each hyperparameter configuration *h*:
     - Train the parameters on the **training** set using *h*.
     - Evaluate the model on the **validation** set.
     - If performance is better than what we got with the best *h* so far (*h**), then save *h* as *h**.
  2. Train a model with *h**, and evaluate its accuracy *A* on the **testing** set. (You can train either on training data, or on training+validation data).

Training/validation/testing sets

To what machine does the reported accuracy *A* correspond?

- Hyperparameter tuning works as follows:

1. For each hyperparameter configuration h:
    - Train the parameters on the **training** set using h.
    - Evaluate the model on the **validation** set.
    - If performance is better than what we got with the best h so far (h^*), then save h as h^*.
2. Train a model with h^*, and evaluate its accuracy A on the **testing** set. (You can train either on training data, or on training+validation data).

Training/validation/testing sets

To what machine does the reported accuracy *A* correspond?

- Hyperparameter tuning works as follows:

1. For each hyperparameter configuration h:
    - Train the parameters on the **training** set using h.
    - Evaluate the model on the **validation** set.
    - If performance is better than what we got with the best h so far (h^*), then save h as h^*.
2. Train a model with h^*, and evaluate its accuracy A on the **testing** set. (You can train either on training data, or on training+validation data).

Cross-validation

- When working with smaller datasets, cross-validation is commonly used so that we can use **all** data for training.

- Suppose we already know the best hyperparameters h^*.

- We partition the data into *k* “folds” of equal sizes.

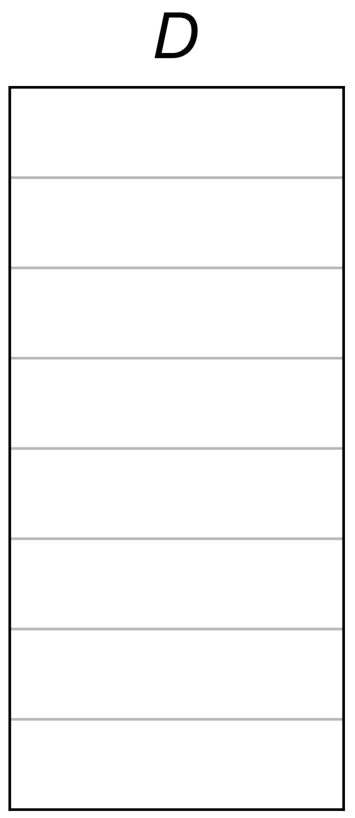
- Over *k* iterations, we train on (*k*-1) folds and test on the remaining fold.

- We then iterate and compute the average accuracy over the *k* testing folds.

Cross-validation

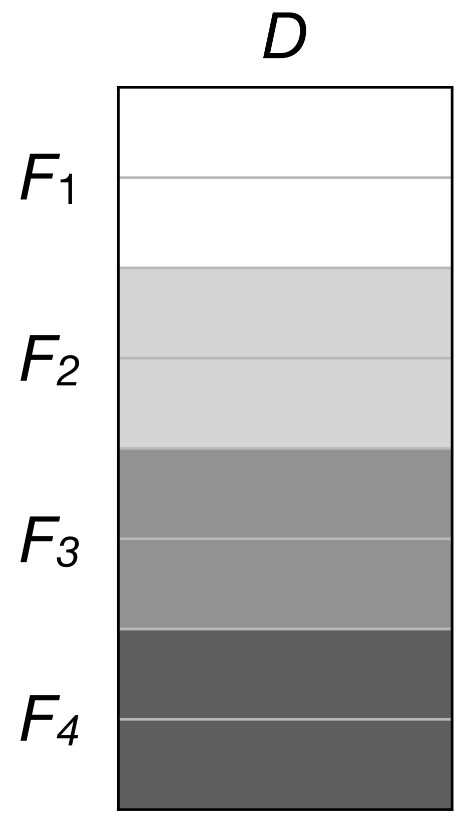
- # *D*=dataset, *k*=# folds, *h*=hyperparameter configuration.

```
CrossValidation ($D$, $k$, $h$):
    Partition $D$ into $k$ folds $F_1, ..., F_k$
    For $i = 1, ..., k$:
        $test = F_i$
        $train = D \setminus F_i$
        Train the model on $train$ using $h$
        $acc[i]$ = Evaluate NN on $test$
    $A$ = Avg[$acc$]
    return $A$
```



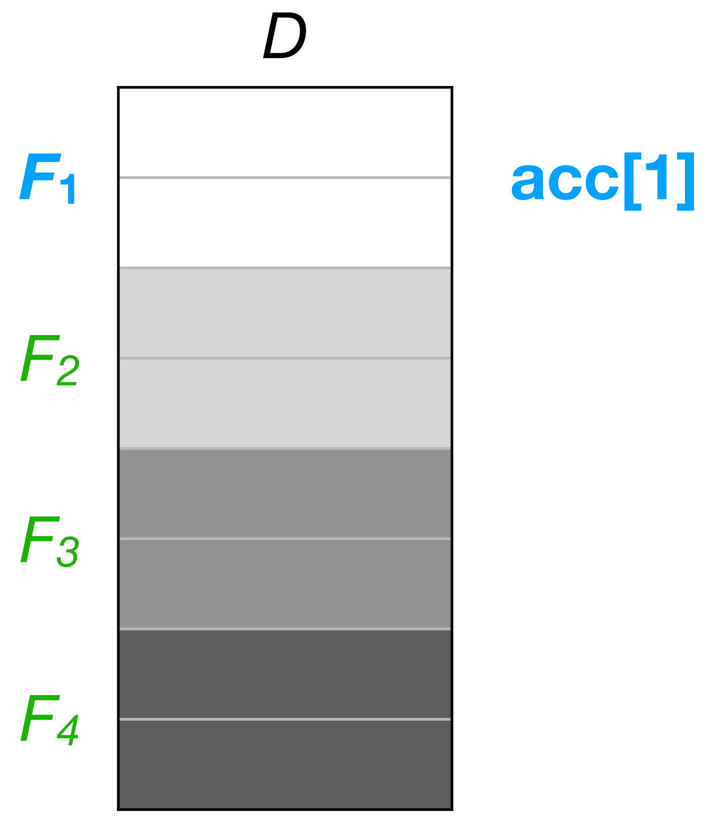
Cross-validation

-
```
# $D$=dataset, $k$=# folds, $h$=hyperparameter configuration.
CrossValidation ($D$, $k$, $h$):
    Partition $D$ into $k$ folds $F_1, ..., F_k$
    For $i = 1, ..., k$:
        $test = F_i$
        $train = D \setminus F_i$
        Train the model on $train$ using $h$
        $acc[i]$ = Evaluate NN on $test$
    $A = \text{Avg}[acc]$
    return $A$
```



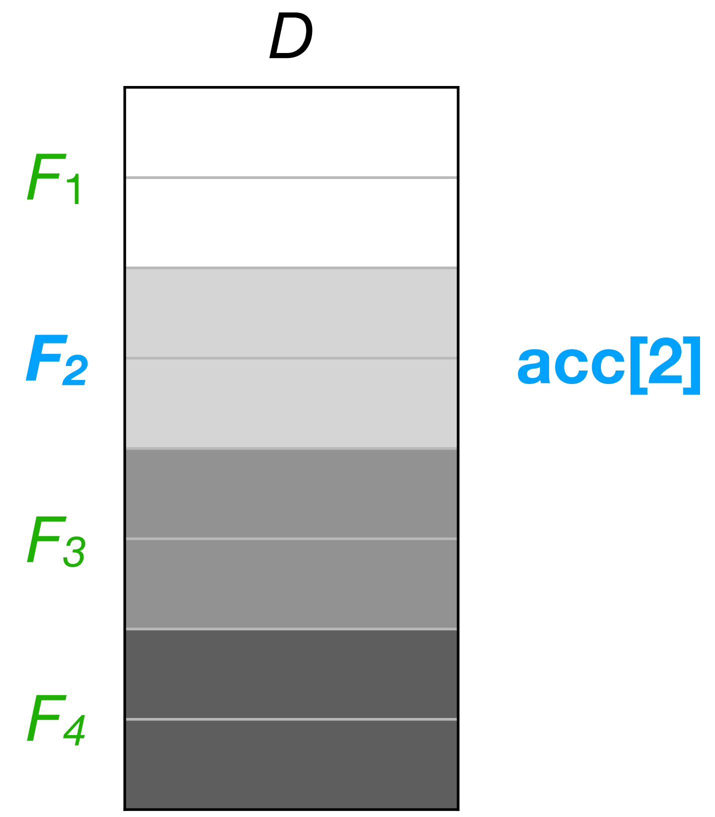
Cross-validation

```
# $D$=dataset, $k$=# folds, $h$=hyperparameter configuration.
CrossValidation ($D$, $k$, $h$):
    Partition $D$ into $k$ folds $F_1, \ldots, F_k$
    For $i = 1, \ldots, k$:
        $test = F_i$
        $train = D \setminus F_i$
        Train the model on $train$ using $h$
        $acc[i]$ = Evaluate NN on $test$
    $A = \text{Avg}[acc]$
    return $A$
```



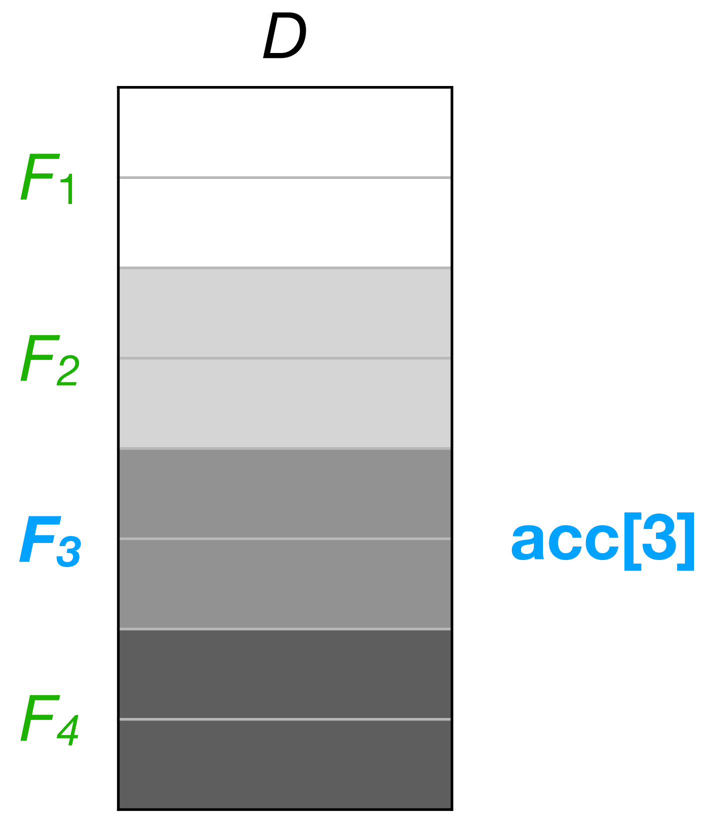
Cross-validation

-
  ```
  # $D$=dataset, $k$=# folds, $h$=hyperparameter configuration.
  CrossValidation ($D$, $k$, $h$):
      Partition $D$ into $k$ folds $F_1$, ..., $F_k$
      For $i$ = 1, ..., $k$:
          $test = F_i$
          $train = D \setminus F_i$
          Train the model on $train$ using $h$
          $acc[i]$ = Evaluate NN on $test$
      $A$ = Avg[$acc$]
      return $A$
  ```



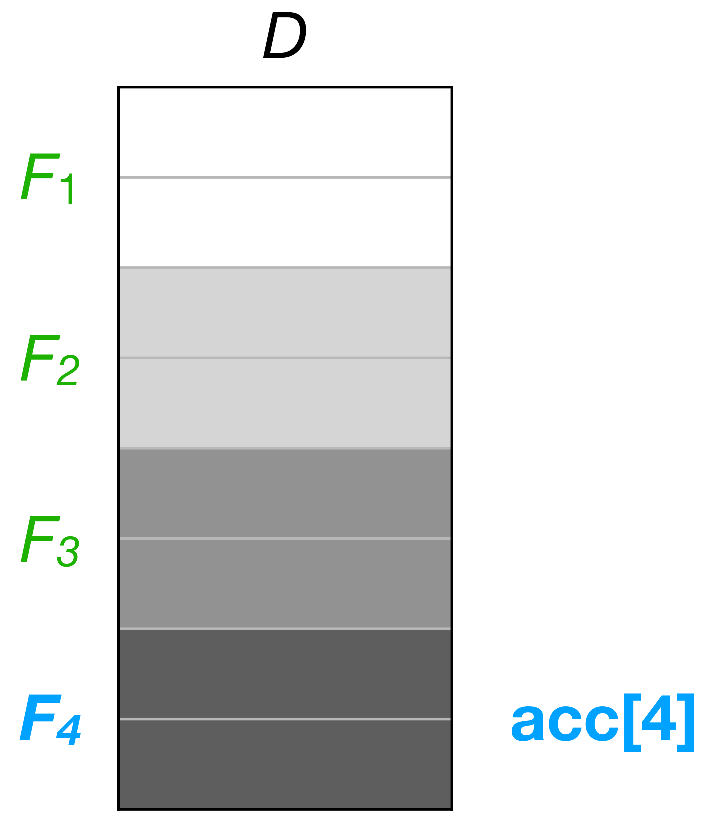
Cross-validation

```
• # $D$=dataset, $k$=# folds, $h$=hyperparameter configuration.
  CrossValidation ($D$, $k$, $h$):
      Partition $D$ into $k$ folds $F_1, \ldots, F_k$
      For $i = 1, \ldots, k$:
          $test = F_i$
          $train = D \setminus F_i$
          Train the model on $train$ using $h$
          $acc[i]$ = Evaluate NN on $test$
      $A = \mathrm{Avg}[acc]$
      return $A$
```



Cross-validation

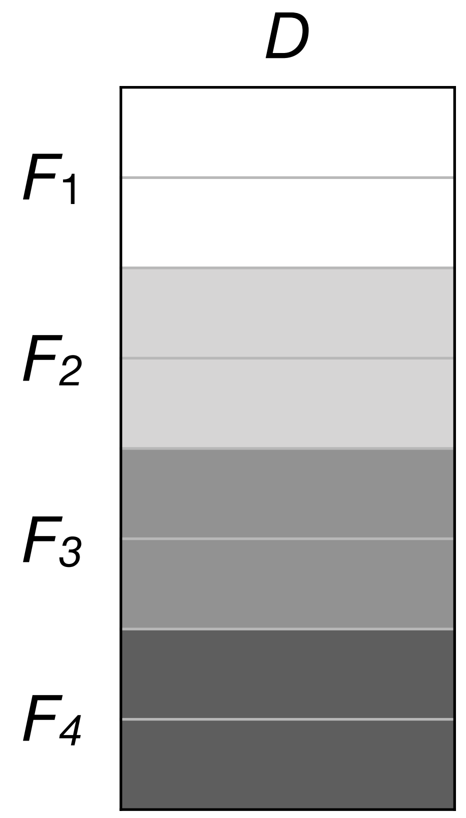
-
  ```
  # $D$=dataset, $k$=# folds, $h$=hyperparameter configuration.
  CrossValidation ($D$, $k$, $h$):
      Partition $D$ into $k$ folds $F_1, \ldots, F_k$
      For $i = 1, \ldots, k$:
          $test = F_i$
          $train = D \setminus F_i$
          Train the model on $train$ using $h$
          $acc[i]$ = Evaluate NN on $test$
      $A = \text{Avg}[acc]$
      return $A$
  ```



Cross-validation

To what machine does the reported accuracy *A* correspond?

```
• # $D$=dataset, $k$=# folds, $h$=hyperparameter configuration.
  CrossValidation ($D$, $k$, $h$):
      Partition $D$ into $k$ folds $F_1, \ldots, F_k$
      For $i = 1, \ldots, k$:
          $test = F_i$
          $train = D \setminus F_i$
          Train the model on $train$ using $h$
          $acc[i]$ = Evaluate NN on $test$
      $A$ = Avg[$acc$]
      return $A$
```

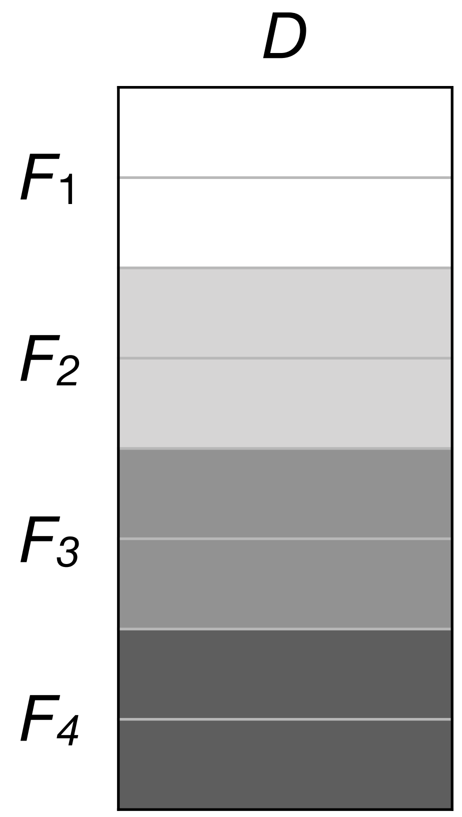


Cross-validation

To what machine does the reported accuracy *A* correspond?

- # *D*=dataset, *k*=# folds, *h*=hyperparameter configuration.

```
CrossValidation ($D$, $k$, $h$):
    Partition $D$ into $k$ folds $F_1, ..., F_k$
    For $i = 1, ..., k$:
        $test = F_i$
        $train = D \setminus F_i$
        Train the model on $train$ using $h$
        $acc[i]$ = Evaluate NN on $test$
    $A = \text{Avg}[acc]$
    return $A$
```



None of them!

Training/validation/testing sets

- Cross-validation does not measure the accuracy of any *single* machine.

- Instead, cross-validation gives the *expected* accuracy of a classifier that is trained on (*k*-1)/*k* of the data.

Training/validation/testing sets

- Cross-validation does not measure the accuracy of any *single* machine.

- Instead, cross-validation gives the *expected* accuracy of a classifier that is trained on (k-1)/k of the data.

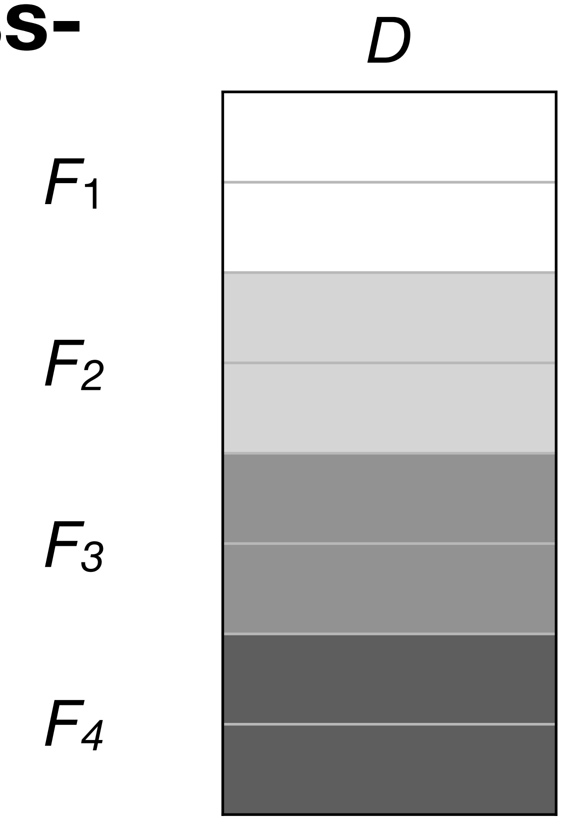
- However, we can train another model M using h^* on the entire dataset, and then report A as its accuracy.

- Since M is trained on more data than any of the cross-validation models, its *expected* accuracy should be >= A.

Double cross-validation

- But how do we find the best hyperparameters h^*?

- A proper approach is to use **double cross-validation**:, i.e., there will be 2 nested for-loops to iterate over "outer" and "inner" folds:

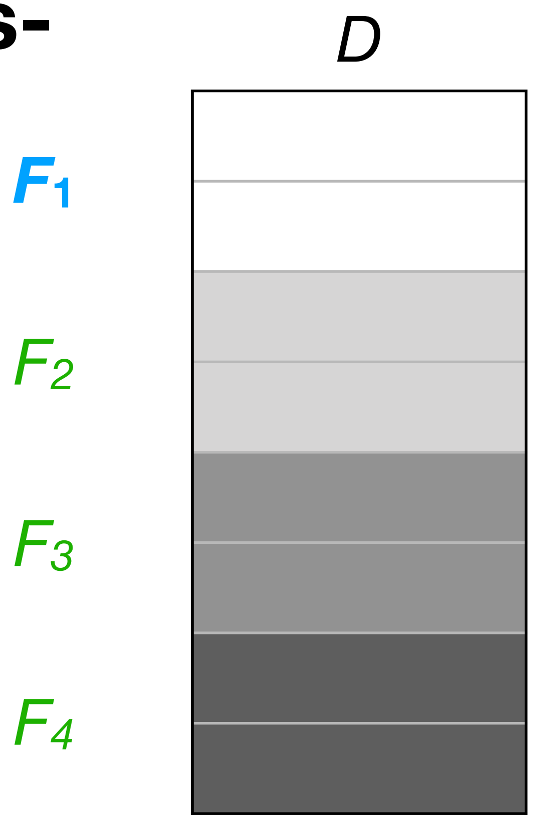


Double cross-validation

- But how do we find the best hyperparameters h^*?

- A proper approach is to use **double cross-validation**:, i.e., there will be 2 nested for-loops to iterate over "outer" and "inner" folds:

  - For each of the k "outer" folds, …

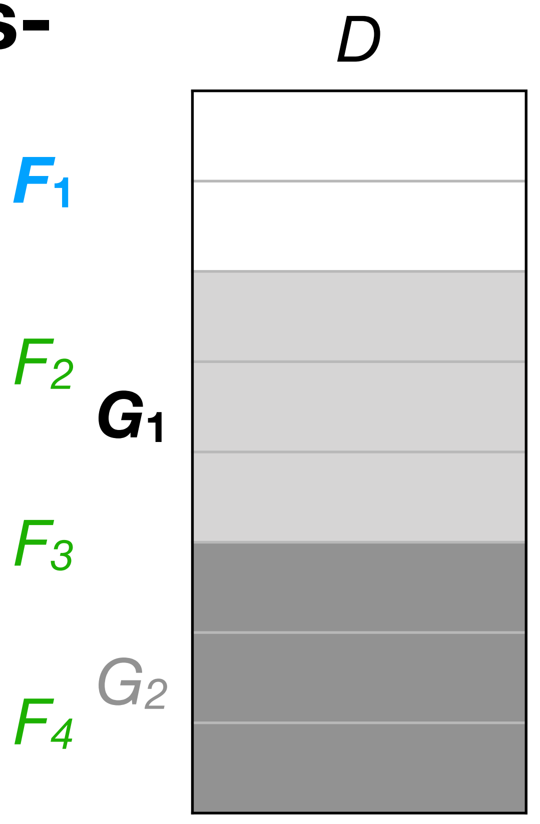


Double cross-validation

- But how do we find the best hyperparameters h^*?

- A proper approach is to use **double cross-validation**:, i.e., there will be 2 nested for-loops to iterate over “outer” and “inner” folds:

    - For each of the k “outer” folds, choose the fold-specific best h^* based on an “inner" cross-validation process.

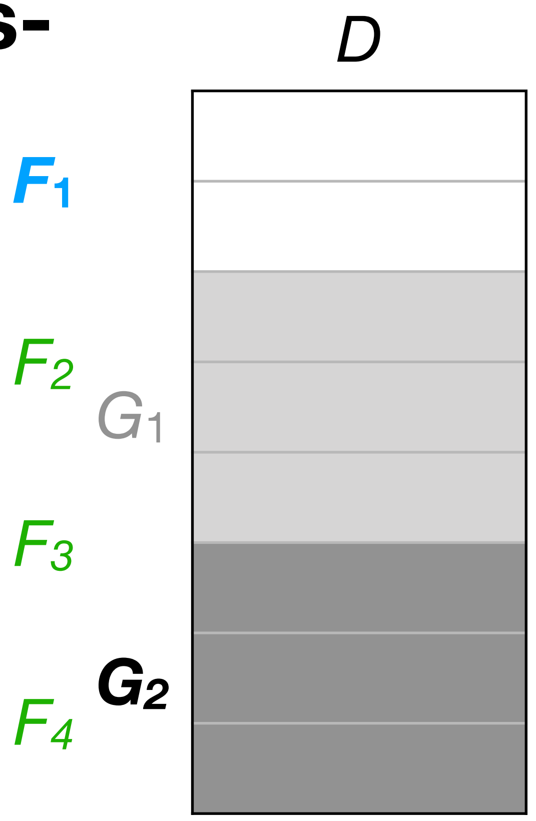


Double cross-validation

- But how do we find the best hyperparameters h^*?

- A proper approach is to use **double cross-validation**:, i.e., there will be 2 nested for-loops to iterate over "outer" and "inner" folds:

  - For each of the k "outer" folds, choose the fold-specific best h^* based on an "inner" cross-validation process.



Double cross-validation

- In contrast to (single) cross-validation, it's not obvious how to finally train a model *M* with accuracy >= *A* (i.e., the accuracy estimated by the procedure).
- One strategy: return an **ensemble** model whose output is the average of the *k* models' predictions…but this is rarely done.
- Double cross-validation is thus more commonly used to validate a new ML approach/architecture, rather than to build a model that is actually "delivered" to a customer.

Dependencies among examples

- In many machine learning settings, the data are not completely independent from each other — they are *linked* in some way.

- Example:

    - Predict multiple grades for each student based on their Canvas clickstream features (# logins, # forum posts, etc.).

Dependencies among examples

- We could partition the data into folds in different ways:
  - We could randomize across all the data.
  - However, if grades are correlated within each student, and if the features can reveal the students' identities, then the training data can leak information about the testing data.

Student	Major	Quiz 1	Quiz 2	Quiz 3
1	CS	45	48	42
2	Math	96	93	93
3	Chemistry	86	86	87
4	Physics	10	30	50

Dependencies among examples

- We could partition the data into folds in different ways:
    - Alternatively, we can **stratify** across students, i.e., no student appears in more than 1 fold.
    - With this partition, the cross-validation accuracy estimates the model's performance on a subject *not* used for training.

Student	Major	Quiz 1	Quiz 2	Quiz 3
1	CS	45	48	42
2	Math	96	93	93
3	Chemistry	86	86	87
4	Physics	10	30	50

Dependencies among examples

- Which strategy to use depends on how you are "marketing" your machine, e.g.:
    - Our machine can predict a *new test score* for any student that the model was trained on; vs…
    - Our machine can predict a test score for a *new student*.

Student	Major	Quiz 1	Quiz 2	Quiz 3
1	CS	45	48	42
2	Math	96	93	93
3	Chemistry	86	86	87
4	Physics	10	30	50

Exercise (from d2l.ai)

- Why is the *K*-fold cross-validation error estimate biased?

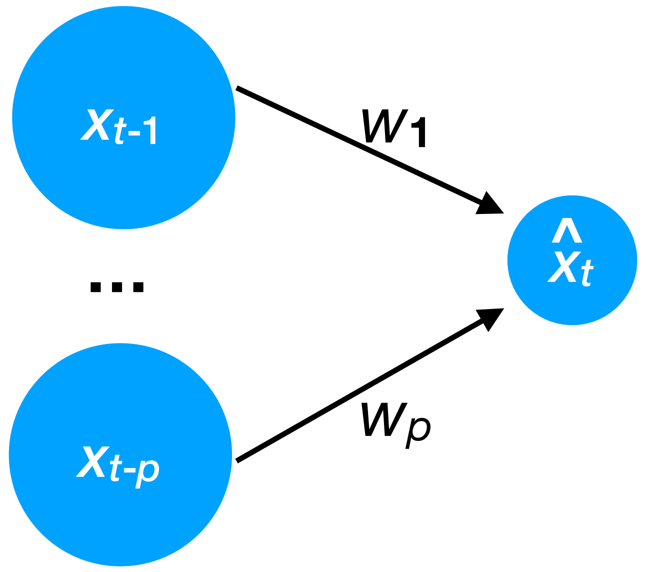
Linear auto-regressive (AR) models

Linear auto-regressive (AR) models

- In some application areas (e.g., economics, finance), we have a time series of values x_1, x_2, …, x_t, but no “labels” y.
- Given the known values of x_1, x_2, …, $x_{t\text{-}1}$, we want to predict the value of x_t.
- A classic example is stock market price prediction.

Linear auto-regressive (AR) models

- In one classic prediction model, we use a fixed length of history (p) to predict the next value x_t:
  - $\hat{x}_t = w_1 x_{t-1} + w_2 x_{t-2} + \ldots + w_p x_{t-p}$
- We can model (and even train) this model using the same 2-layer neural network as before:



Auto-regression

- The essence of auto-regression is that we are using the past to predict the next future event.
- We can apply this recursively to predict infinitely into the future.
- Example for p=2, assuming we already know x_1, x_2:
  - $\hat{x}_3 = w_1 x_2 + w_2 x_1$

Auto-regression

- The essence of auto-regression is that we are using the past to predict the next future event.
- We can apply this recursively to predict infinitely into the future.
- Example for p=2, assuming we already know x_1, x_2:
  - $\hat{x}_3 = w_1 x_2 + w_2 x_1$
  - $\hat{x}_4 = w_1 x_3 + w_2 x_2$

Auto-regression

- The essence of auto-regression is that we are using the past to predict the next future event.

- We can apply this recursively to predict infinitely into the future.

- Example for $p=2$, assuming we already know x_1, x_2:
  - $\hat{x}_3 = w_1 x_2 + w_2 x_1$
  - $\hat{x}_4 = w_1 x_3 + w_2 x_2$
  - $\hat{x}_5 = w_1 x_4 + w_2 x_3$
  - *...*

Exercise

- For w_1=0.4, w_2=-0.5, x_1=0, and x_2=2, what are the predictions for x_3, x_4, and x_5?

Exercise

- For w_1=0.4, w_2=-0.5, x_1=0, and x_2=2, what are the predictions for x_3, x_4, and x_5?
  - $\hat{x}_3$ = (0.4)2 + (-0.5)0 = 0.8
  - $\hat{x}_4$ = (0.4)0.8 + (-0.5)2 = -0.68
  - $\hat{x}_5$ = (0.4)(-.68) + (-0.5)(0.8) = -0.672

Multivariate auto-regression

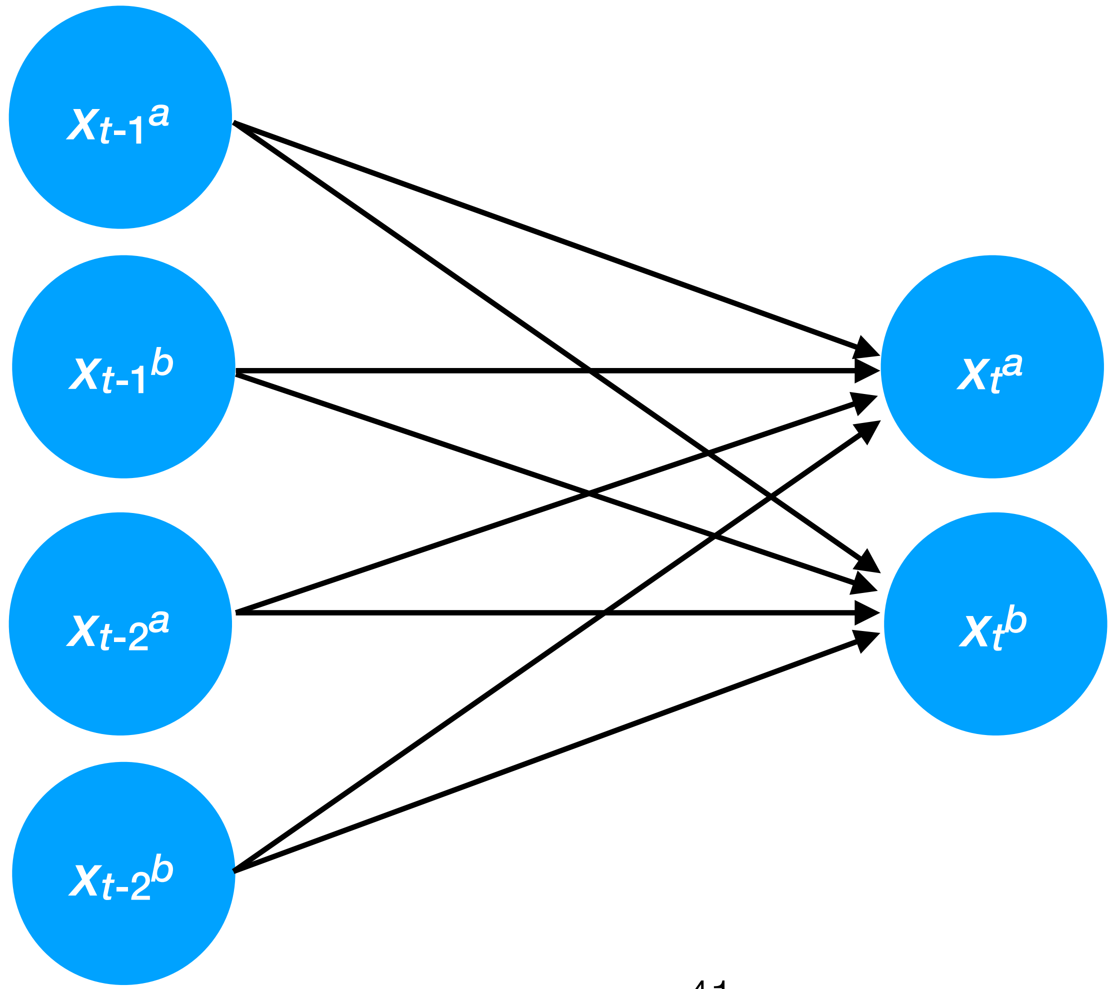
- The value $\mathbf{x}_t$ of each time-step can also be a vector.
- In this case, each weight is a ?

Multivariate auto-regression

- The value $\mathbf{x}_t$ of each time-step can also be a vector.
- In this case, each weight is a matrix $\mathbf{W}$.
- $\hat{\mathbf{x}}_t = \mathbf{W}^{(1)} \mathbf{x}_{t-1} + \ldots + \mathbf{W}^{(p)} \mathbf{x}_{t-p}$

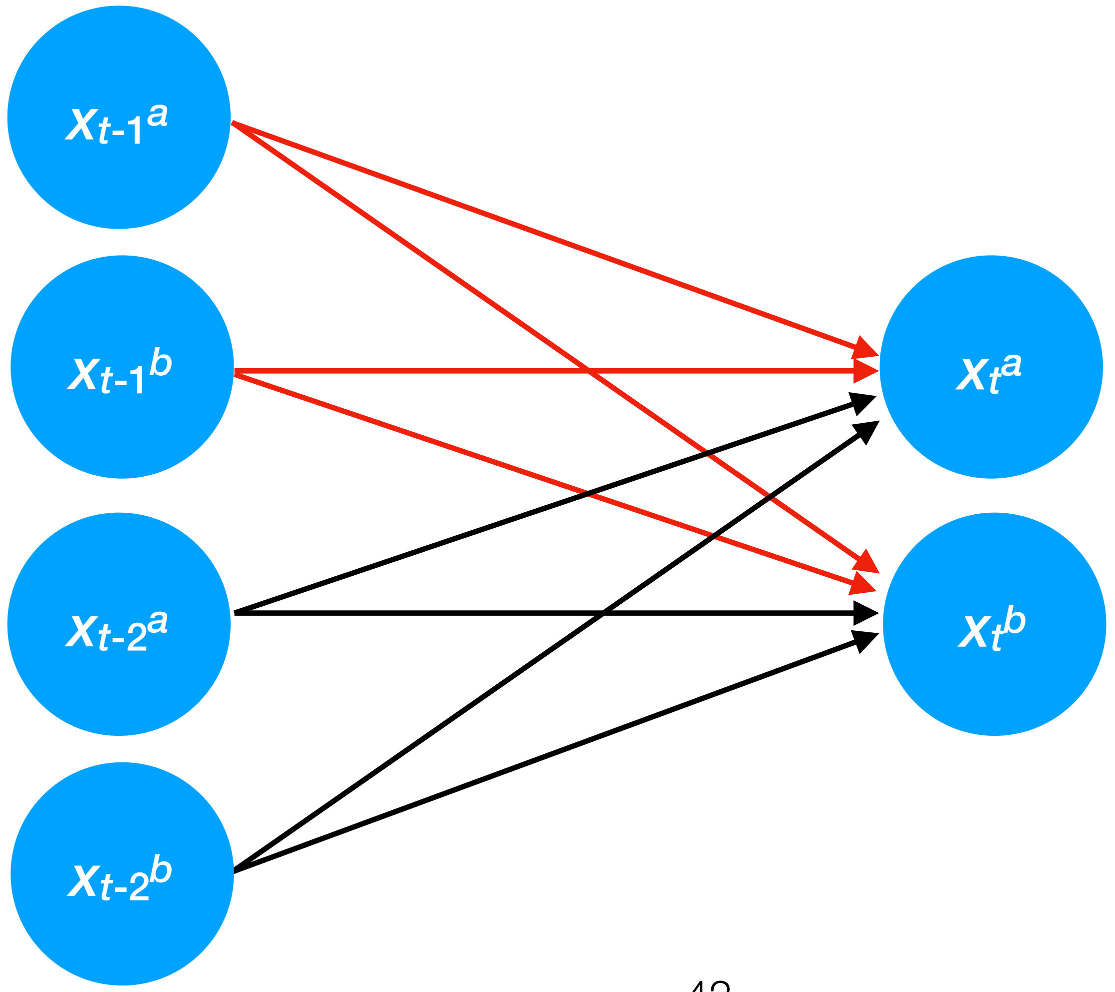
Multivariate auto-regression

- Suppose each observation $\mathbf{x}_t$ has 2 components (x_t^a, x_t^b), and that p=2.
- Here is the corresponding neural network:



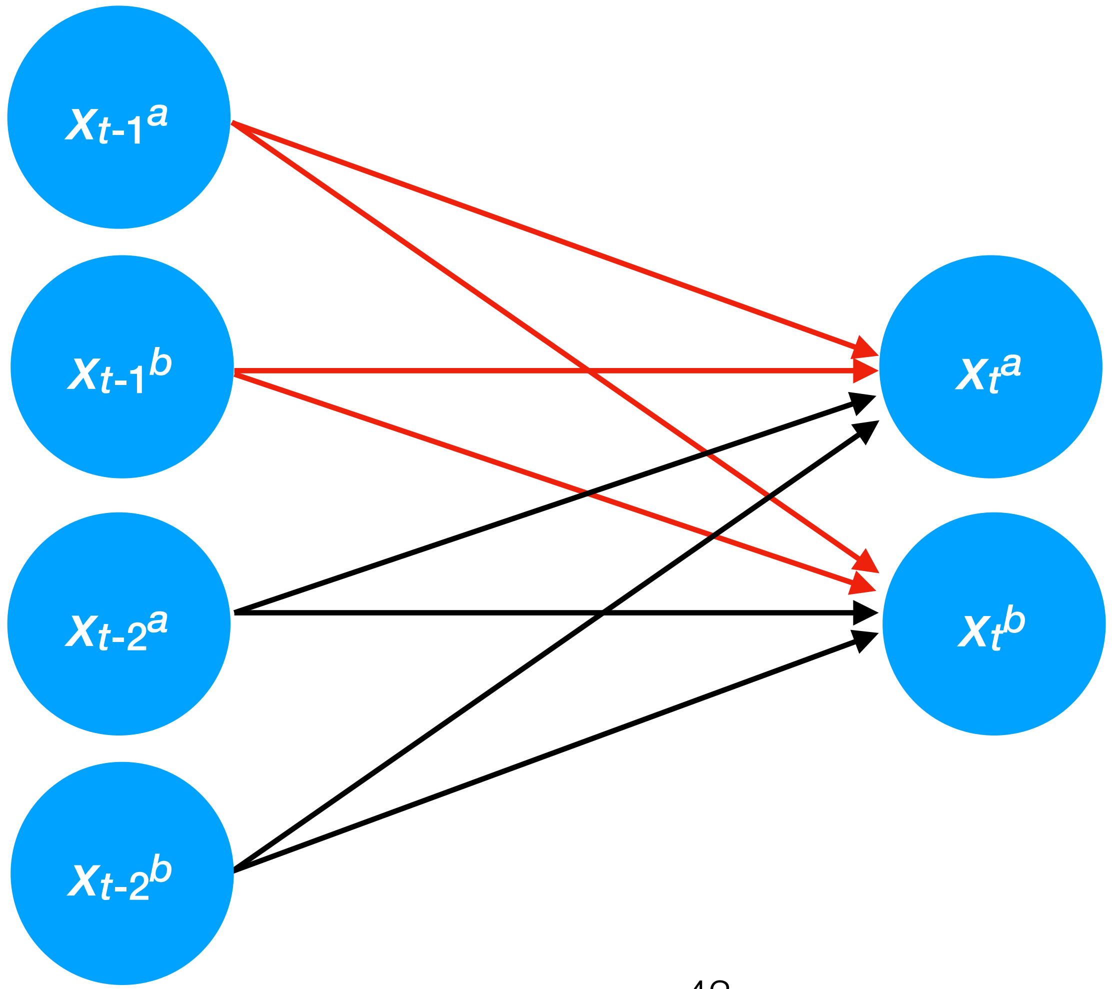
Exercise

- Recall: $\hat{\mathbf{x}}_t = \mathbf{W}^{(1)}\, \mathbf{x}_{t\text{-}1} + \ldots + \mathbf{W}^{(p)}\, \mathbf{x}_{t\text{-}p}$
- To which matrix ($\mathbf{W}^{(1)}$, $\mathbf{W}^{(2)}$, or neither) do the first 4 edges correspond?



Exercise

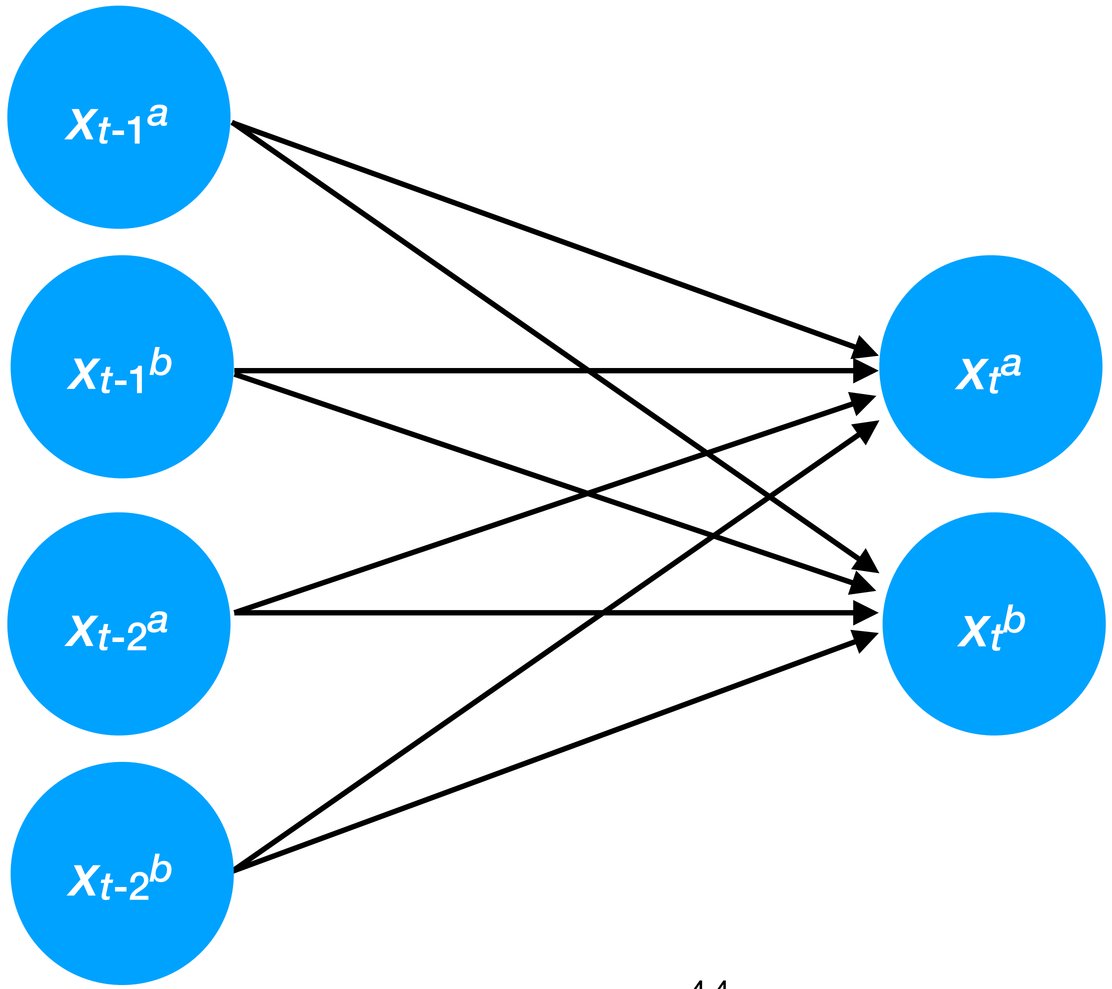
- Recall: $\hat{\mathbf{x}}_t = \mathbf{W}^{(1)}\, \mathbf{x}_{t\text{-}1} + \ldots + \mathbf{W}^{(p)}\, \mathbf{x}_{t\text{-}p}$
- To which matrix ($\mathbf{W}^{(1)}$, $\mathbf{W}^{(2)}$, or neither) do the first 4 edges correspond? $\mathbf{W}^{(1)}$.



Multivariate auto-regression

- We can alternatively represent this network with just a single matrix of weights **W** if we "stack" the inputs:

- $\hat{\mathbf{x}}_t = \mathbf{W}\, [\mathbf{x}_{t-1}^\top ; \ldots ; \mathbf{x}_{t-p}^\top]^\top$



Auto-regression in deep learning

- Auto-regression is used frequently in deep learning, especially for machine translation and text generation (e.g., ChatGPT).

Stochastic gradient descent

Gradient descent

- With gradient descent, we only update the weights after scanning the *entire* training set.
    - This is slow.
- If the training set contains 20K examples, then the gradient is an *average* over 20K images.
    - How much would the gradient really change if we just used, say, 10K images? 5K images? 128 images?

$$\nabla_{\mathbf{w}} f_{\mathrm{MSE}}(\mathbf{y}, \hat{\mathbf{y}}; \mathbf{w}) = \frac{1}{n}\mathbf{X}(\mathbf{X}^\top \mathbf{w} - \mathbf{y})$$

Average over entire training set.

Stochastic gradient descent

- This is the idea behind **stochastic gradient descent** (SGD):
  - Randomly sample a small ($\ll n$) **mini-batch** (or sometimes just **batch**) of training examples.
  - Estimate the gradient on just the mini-batch.
  - Update weights based on *mini-batch* gradient estimate.
  - Repeat.

Stochastic gradient descent

- In practice, SGD is usually conducted over multiple epochs.
  - An **epoch** is a single pass through the entire training set.
- Procedure:
  1. Let $\tilde{n} \ll n$ equal the size of the mini-batch.

Stochastic gradient descent

- In practice, SGD is usually conducted over multiple epochs.
    - An **epoch** is a single pass through the entire training set.

- Procedure:
    1. Let $\tilde{n} \ll n$ equal the size of the mini-batch.
    2. Randomize the order of the examples in the training set.

Stochastic gradient descent

- In practice, SGD is usually conducted over multiple epochs.
    - An **epoch** is a single pass through the entire training set.
- Procedure:
    1. Let $\tilde{n} \ll n$ equal the size of the mini-batch.
    2. Randomize the order of the examples in the training set.
    3. For *e* = 0 to numEpochs:

Stochastic gradient descent

- In practice, SGD is usually conducted over multiple epochs.
  - An **epoch** is a single pass through the entire training set.
- Procedure:
  1. Let $\tilde{n} \ll n$ equal the size of the mini-batch.
  2. Randomize the order of the examples in the training set.
  3. For *e* = 0 to numEpochs:
     I. For *i* = 0 to $(\lceil n/\tilde{n} \rceil - 1)$ (one epoch):

Stochastic gradient descent

- In practice, SGD is usually conducted over multiple epochs.
    - An **epoch** is a single pass through the entire training set.
- Procedure:

```
1. Let $\tilde{n} \ll n$ equal the size of the mini-batch.
2. Randomize the order of the examples in the training set.
3. For $e$ = 0 to numEpochs:
    I. For $i$ = 0 to $(\lceil n/\tilde{n} \rceil - 1)$ (one epoch):
        A. Select a mini-batch $\mathcal{J}$ containing the next $\tilde{n}$ examples.
```

Stochastic gradient descent

- In practice, SGD is usually conducted over multiple epochs.
  - An **epoch** is a single pass through the entire training set.
- Procedure:
  1. Let $\tilde{n} \ll n$ equal the size of the mini-batch.
  2. Randomize the order of the examples in the training set.
  3. For *e* = 0 to numEpochs:
     I. For *i* = 0 to $(\lceil n/\tilde{n} \rceil - 1)$ (one epoch):
        A. Select a mini-batch $\mathcal{J}$ containing the next $\tilde{n}$ examples.
        B. Compute the gradient on this mini-batch: $\frac{1}{\tilde{n}} \sum_{i \in \mathcal{J}} \nabla_{\mathbf{W}} f(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}; \mathbf{W})$

Stochastic gradient descent

- In practice, SGD is usually conducted over multiple epochs.
    - An **epoch** is a single pass through the entire training set.

- Procedure:
    1. Let $\tilde{n} \ll n$ equal the size of the mini-batch.
    2. Randomize the order of the examples in the training set.
    3. For *e* = 0 to numEpochs:
        I. For *i* = 0 to $(\lceil n/\tilde{n} \rceil - 1)$ (one epoch):
            A. Select a mini-batch $\mathcal{J}$ containing the next $\tilde{n}$ examples.
            B. Compute the gradient on this mini-batch: $\frac{1}{\tilde{n}} \sum_{i \in \mathcal{J}} \nabla_{\mathbf{W}} f(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}; \mathbf{W})$
            C. Update the weights based on the current mini-batch gradient.

SGD versus GD: example

- Suppose our training set contains *n*=8 examples.

- Here is how *regular* gradient descent would proceed:
  - **Initialize weights $\mathbf{w}^{(0)}$ to random values.**

Training examples
1
2
3
4
5
6
7
8

SGD versus GD: example

- Suppose our training set contains *n*=8 examples.
- Here is how *regular* gradient descent would proceed:
  - Initialize weights $\mathbf{w}^{(0)}$ to random values.
  - For each round:
    - **Compute gradient on all *n* examples.**

Training examples
1
2
3
4
5
6
7
8

SGD versus GD: example

- Suppose our training set contains n=8 examples.
- Here is how *regular* gradient descent would proceed:
  - Initialize weights $\mathbf{w}^{(0)}$ to random values.
  - For each round:
    - Compute gradient on all n examples.
    - **Update weights**: $\mathbf{w}^{(t+1)} \longleftarrow \mathbf{w}^{(t)} - \epsilon \nabla_{\mathbf{w}} f$

Training examples
1
2
3
4
5
6
7
8

SGD versus GD: example

- Suppose our training set contains n=8 examples.
- Here is how *regular* gradient descent would proceed:
  - Initialize weights $\mathbf{w}^{(0)}$ to random values.
  - For each round:
    - **Compute gradient on all *n* examples.**
    - Update weights: $\mathbf{w}^{(t+1)} \longleftarrow \mathbf{w}^{(t)} - \epsilon \nabla_{\mathbf{w}} f$

Training examples
1
2
3
4
5
6
7
8

SGD versus GD: example

- Suppose our training set contains n=8 examples.
- Here is how *regular* gradient descent would proceed:
  - Initialize weights $\mathbf{w}^{(0)}$ to random values.
  - For each round:
    - Compute gradient on all n examples.
    - **Update weights**: $\mathbf{w}^{(t+1)} \longleftarrow \mathbf{w}^{(t)} - \epsilon \nabla_{\mathbf{w}} f$

Training examples
1
2
3
4
5
6
7
8

SGD versus GD: example

- Suppose our training set contains *n*=8 examples with $\tilde{n}$ = 2.
- Here is how *stochastic* gradient descent would proceed:
  - **Initialize weights $\mathbf{w}^{(0)}$ to random values.**

Training examples
1
2
3
4
5
6
7
8

SGD versus GD: example

- Suppose our training set contains *n*=8 examples with $\tilde{n}$ = 2.
- Here is how *stochastic* gradient descent would proceed:
  - Initialize weights $\mathbf{w}^{(0)}$ to random values.
  - **Randomize the order of the training data.**

Training examples
4
1
3
5
7
6
8
2

SGD versus GD: example

- Suppose our training set contains n=8 examples with $\tilde{n}$ = 2.
- Here is how *stochastic* gradient descent would proceed:
  - Initialize weights $\mathbf{w}^{(0)}$ to random values.
  - Randomize the order of the training data.
  - For each epoch (e=1, …, E): e=1
    - For each round (r=1, …, $\lceil n/\tilde{n} \rceil$):
      - **Compute gradient on next $\tilde{n}$ examples.**

Training examples
4
1
3
5
7
6
8
2

SGD versus GD: example

- Suppose our training set contains n=8 examples with $\tilde{n}$ = 2.
- Here is how *stochastic* gradient descent would proceed:
  - Initialize weights $\mathbf{w}^{(0)}$ to random values.
  - Randomize the order of the training data.
  - For each epoch (e=1, …, E): e=1
    - For each round (r=1, …, $\lceil n/\tilde{n} \rceil$):
      - Compute gradient on next $\tilde{n}$ examples.
      - **Update weights**: $\mathbf{w}^{(t+1)} \longleftarrow \mathbf{w}^{(t)} - \epsilon \widetilde{\nabla_{\mathbf{w}} f}$

Training examples
4
1
3
5
7
6
8
2

SGD versus GD: example

- Suppose our training set contains *n*=8 examples with $\tilde{n}$ = 2.
- Here is how *stochastic* gradient descent would proceed:
  - Initialize weights $\mathbf{w}^{(0)}$ to random values.
  - Randomize the order of the training data.
  - For each epoch (*e*=1, …, *E*): e=1
    - For each round (*r*=1, …, $\lceil n/\tilde{n} \rceil$):
      - **Compute gradient on next $\tilde{n}$ examples.**
      - Update weights: $\mathbf{w}^{(t+1)} \longleftarrow \mathbf{w}^{(t)} - \epsilon \nabla_{\mathbf{w}} \tilde{f}$

Training examples
4
1
3
5
7
6
8
2

SGD versus GD: example

- Suppose our training set contains *n*=8 examples with $\tilde{n}$ = 2.
- Here is how *stochastic* gradient descent would proceed:
  - Initialize weights $\mathbf{w}^{(0)}$ to random values.
  - Randomize the order of the training data.
  - For each epoch (*e*=1, …, *E*): **e=1**
    - For each round (*r*=1, …, $\lceil n/\tilde{n} \rceil$):
      - Compute gradient on next $\tilde{n}$ examples.
      - **Update weights**: $\mathbf{w}^{(t+1)} \longleftarrow \mathbf{w}^{(t)} - \epsilon \tilde{\nabla}_{\mathbf{w}} f$

Training examples
4
1
3
5
7
6
8
2

SGD versus GD: example

- Suppose our training set contains n=8 examples with $\tilde{n}$ = 2.
- Here is how *stochastic* gradient descent would proceed:
  - Initialize weights $\mathbf{w}^{(0)}$ to random values.
  - Randomize the order of the training data.
  - For each epoch (e=1, …, E): e=1
    - For each round (r=1, …, $\lceil n/\tilde{n} \rceil$):
      - **Compute gradient on next $\tilde{n}$ examples.**
      - Update weights: $\mathbf{w}^{(t+1)} \longleftarrow \mathbf{w}^{(t)} - \epsilon \tilde{\nabla_{\mathbf{w}} f}$

Training examples
4
1
3
5
7
6
8
2

SGD versus GD: example

- Suppose our training set contains n=8 examples with $\tilde{n} = 2$.
- Here is how *stochastic* gradient descent would proceed:
  - Initialize weights $\mathbf{w}^{(0)}$ to random values.
  - Randomize the order of the training data.
  - For each epoch (e=1, …, E): e=1
    - For each round (r=1, …, $\lceil n/\tilde{n} \rceil$):
      - Compute gradient on next $\tilde{n}$ examples.
      - **Update weights**: $\mathbf{w}^{(t+1)} \longleftarrow \mathbf{w}^{(t)} - \epsilon \widetilde{\nabla_{\mathbf{w}} f}$

Training examples
4
1
3
5
7
6
8
2

SGD versus GD: example

- Suppose our training set contains n=8 examples with $\tilde{n}$ = 2.
- Here is how *stochastic* gradient descent would proceed:
  - Initialize weights $\mathbf{w}^{(0)}$ to random values.
  - Randomize the order of the training data.
  - For each epoch (e=1, …, E): e=1
    - For each round (r=1, …, $\lceil n/\tilde{n} \rceil$):
      - **Compute gradient on next $\tilde{n}$ examples.**
      - Update weights: $\mathbf{w}^{(t+1)} \longleftarrow \mathbf{w}^{(t)} - \epsilon \tilde{\nabla}_{\mathbf{w}} f$

Training examples
4
1
3
5
7
6
8
2

SGD versus GD: example

- Suppose our training set contains *n*=8 examples with $\tilde{n}$ = 2.
- Here is how *stochastic* gradient descent would proceed:
  - Initialize weights $\mathbf{w}^{(0)}$ to random values.
  - Randomize the order of the training data.
  - For each epoch (*e*=1, …, *E*): e=1
    - For each round (*r*=1, …, $\lceil n/\tilde{n} \rceil$):
      - Compute gradient on next $\tilde{n}$ examples.
      - **Update weights**: $\mathbf{w}^{(t+1)} \longleftarrow \mathbf{w}^{(t)} - \epsilon \tilde{\nabla}_{\mathbf{w}} f$

Training examples
4
1
3
5
7
6
8
2

SGD versus GD: example

- Suppose our training set contains n=8 examples with $\tilde{n}$ = 2.
- Here is how *stochastic* gradient descent would proceed:
  - Initialize weights $\mathbf{w}^{(0)}$ to random values.
  - Randomize the order of the training data.
  - For each epoch (e=1, …, E): e=2
    - For each round (r=1, …, $\lceil n/\tilde{n} \rceil$):
      - **Compute gradient on next $\tilde{n}$ examples.**
      - Update weights: $\mathbf{w}^{(t+1)} \longleftarrow \mathbf{w}^{(t)} - \epsilon \tilde{\nabla}_{\mathbf{w}} f$

Training examples
4
1
3
5
7
6
8
2

SGD versus GD: example

- Suppose our training set contains n=8 examples with $\tilde{n}$ = 2.
- Here is how *stochastic* gradient descent would proceed:
  - Initialize weights $\mathbf{w}^{(0)}$ to random values.
  - Randomize the order of the training data.
  - For each epoch (e=1, …, E): e=2
    - For each round (r=1, …, $\lceil n/\tilde{n} \rceil$):
      - Compute gradient on next $\tilde{n}$ examples.
      - Update weights: $\mathbf{w}^{(t+1)} \longleftarrow \mathbf{w}^{(t)} - \epsilon \widetilde{\nabla_{\mathbf{w}} f}$

    ...

Training examples
4
1
3
5
7
6
8
2

Stochastic gradient descent

- Despite “noise” (statistical inaccuracy) in the mini-batch gradient estimates, we will still converge to local minimum.

- Training can be much faster than regular gradient descent because we adjust the weights *many times* per epoch.

SGD: learning rates

- With SGD, our learning rate ϵ needs to be **annealed** (reduced slowly over time) to guarantee convergence.
  - Otherwise we might just oscillate forever in weight space.
- Necessary conditions:

$$\lim_{T\to\infty} \sum_{t=1}^{T} |\epsilon_t|^2 < \infty$$

Not too big: sum of squared learning rates converges.

SGD: learning rates

- With SGD, our learning rate ϵ needs to be **annealed** (reduced slowly over time) to guarantee convergence.
  - Otherwise we might just oscillate forever in weight space.
- Necessary conditions:

$$\lim_{T\to\infty} \sum_{t=1}^{T} |\epsilon_t|^2 < \infty \qquad \lim_{T\to\infty} \sum_{t=1}^{T} |\epsilon_t| = \infty$$

Not too small: sum of absolute learning rates grows to infinity.

SGD: learning rates

- One common learning rate "schedule" is to multiply ϵ by $c \in (0, 1)$ every *k* rounds.
  - This is called **exponential decay**.
- Another possibility (which avoids the issue) is to set the number of epochs *T* to a finite number.
  - SGD may not fully converge, but the machine might still perform well.
- There are many other strategies.

L_2 Regularization

Regularization

- The larger the coefficients (weights) **w** are allowed to be, the more the neural network can overfit.
- If we “encourage” the weights to be small, we can reduce overfitting.
- This is a form of **regularization** — any practice designed to improve the machine’s ability to **generalize** to new data.

Regularization

- One of the simplest and oldest regularization techniques is to *penalize* large weights in the cost function.

- The “unregularized” f_{MSE} is:

$$f_{\text{MSE}}(\mathbf{w}) = \frac{1}{2n}\sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2$$

Regularization

- One of the simplest and oldest regularization techniques is to *penalize* large weights in the cost function.

- The "unregularized" f_{MSE} is:

$$f_{\text{MSE}}(\mathbf{w}) = \frac{1}{2n}\sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2$$

- The L_2-regularized f_{MSE} becomes:

$$f_{\text{MSE}}(\mathbf{w}) = \frac{1}{2n}\sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2 + \frac{\alpha}{2n}\mathbf{w}^\top\mathbf{w}$$

Regularization

- The closed-form solution for the optimal weight vector **w** in L_2-regularized linear regression is then:

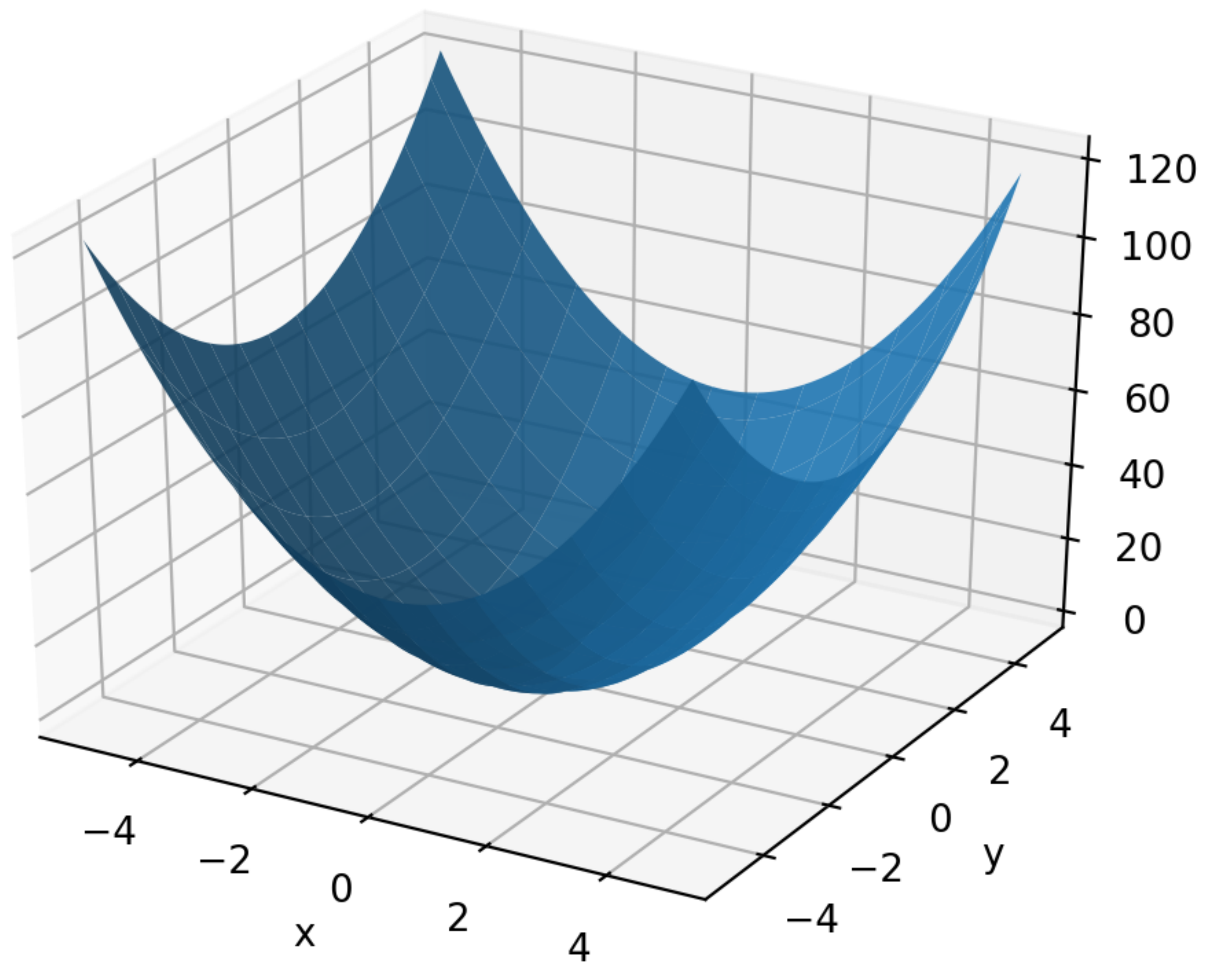
$$\mathbf{w} = (\mathbf{X}\mathbf{X}^\top + \alpha\mathbf{I})^{-1}\mathbf{X}\mathbf{y}$$

- Note that the inverse of the matrix in parentheses is now guaranteed to be invertible — a handy bonus.

Optimization of ML models

Optimization of ML models

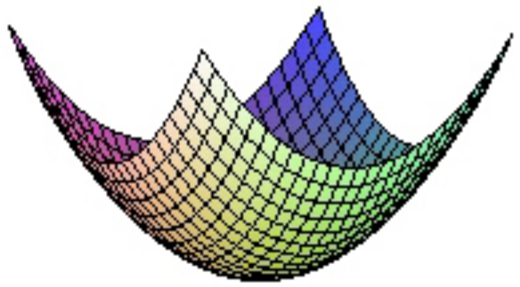
- With linear regression, the cost function f_{MSE} has a single local minimum w.r.t. the weights **w**:



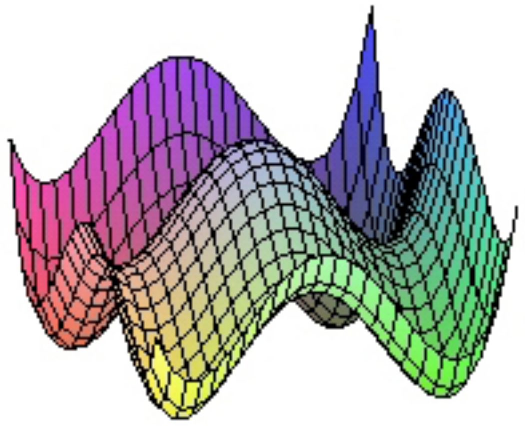
- As long as our learning rate is small enough, we will eventually find **the optimal w**.

Convex ML models

- Linear regression has a loss function that is **convex**.
- With a convex function *f*, *every local minimum is also a global minimum*.



convex



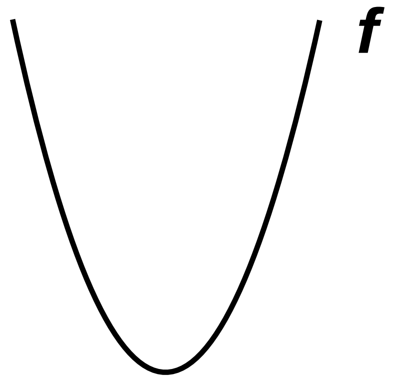
non-convex

- Convex functions are ideal for conducting gradient descent.

https://plus.maths.org/content/convexity

Convexity in 1-d

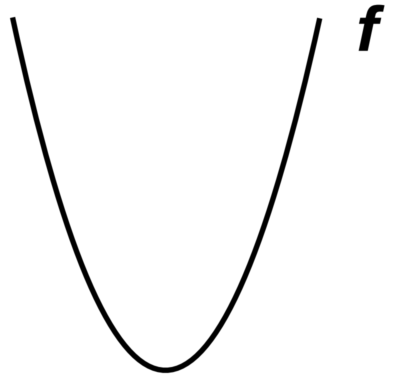
- How can we tell if a 1-d function *f* is convex?



- What property of the slope of *f* ensures there is only one local minimum?

Convexity in 1-d

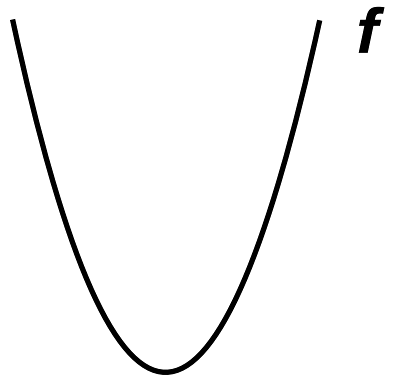
- How can we tell if a 1-d function *f* is convex?



- What property of the slope of *f* ensures there is only one local minimum?
    - From left to right, the slope of *f never decreases*.

Convexity in 1-d

- How can we tell if a 1-d function *f* is convex?

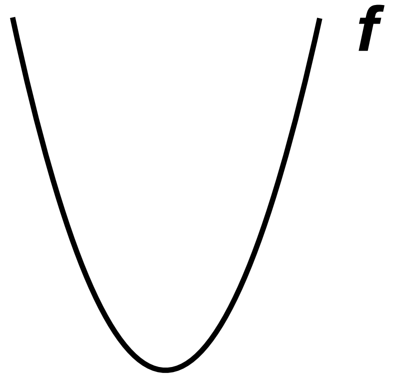


- What property of the slope of *f* ensures there is only one local minimum?

  - From left to right, the slope of *f never decreases*.
    ==> the derivative of the slope is always non-negative.

Convexity in 1-d

- How can we tell if a 1-d function *f* is convex?



- What property of the slope of *f* ensures there is only one local minimum?

  - From left to right, the slope of *f* *never decreases*.
    ==> the derivative of the slope is always non-negative.
    ==> the second derivative of *f* is always non-negative.

Convexity in higher dimensions

- For higher-dimensional *f*, convexity is determined by the the Hessian of *f*.

$$\mathrm{H}[f] = \left[\begin{array}{ccc} \frac{\partial^2 f}{\partial x_1 \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_m} \\ \ldots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_m \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_m \partial x_m} \end{array} \right]$$

- For $f : \mathbb{R}^m \rightarrow \mathbb{R}$, *f* is convex if the Hessian matrix is positive semi-definite for *every* input **x**.

Positive semi-definite

- Positive semi-definite is the matrix analog of being “non-negative”.

- A real symmetric matrix **A** is **positive semi-definite (PSD)** if (equivalent conditions):

Positive semi-definite

- Positive semi-definite is the matrix analog of being “non-negative”.

- A real symmetric matrix **A** is **positive semi-definite (PSD)** if (equivalent conditions):
  - All its eigenvalues are ≥0.
    - In particular, if **A** happens to be diagonal, then **A** is PSD if its eigenvalues are the diagonal elements.

Positive semi-definite

- Positive semi-definite is the matrix analog of being "non-negative".

- A real symmetric matrix **A** is **positive semi-definite (PSD)** if (equivalent conditions):
  - All its eigenvalues are ≥ 0.
    - In particular, if **A** happens to be diagonal, then **A** is PSD if its eigenvalues are the diagonal elements.
  - For every vector **v**: $\mathbf{v}^T\mathbf{A}\mathbf{v} \geq 0$
    - Therefore: If there exists *any* vector **v** such that $\mathbf{v}^T\mathbf{A}\mathbf{v} < 0$, then **A** is *not* PSD.

Example

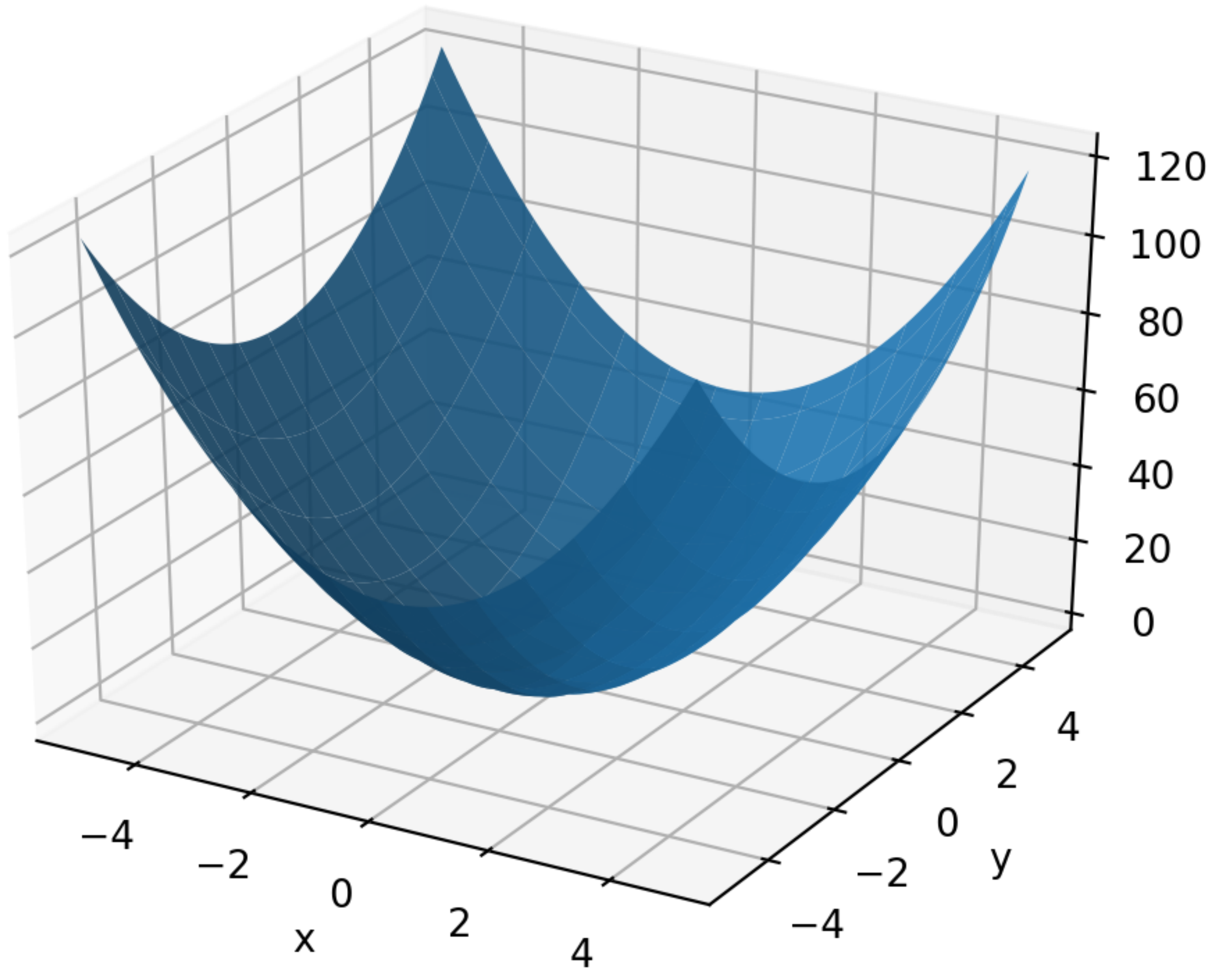
- Suppose *f*(*x*, *y*) = $3x^2 + 2y^2 - 2$.
- Then the first derivatives are: $\frac{\partial f}{\partial x} = 6x \quad \frac{\partial f}{\partial y} = 4y$
- The Hessian matrix is therefore:

$$\mathbf{H} = \left[\begin{array}{cc} \frac{\partial^2 f}{\partial x \partial x} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial y \partial x} & \frac{\partial^2 f}{\partial y \partial y} \end{array} \right] = \left[\begin{array}{cc} 6 & 0 \\ 0 & 4 \end{array} \right]$$

- Notice that **H** for this *f* does not depend on (*x*,*y*).
- Also, **H** is a diagonal matrix (with 6 and 4 on the diagonal). Hence, the eigenvalues are just 6 and 4. Since they are both non-negative, then *f* is convex.

Example

- Graph of $f(x, y) = 3x^2 + 2y^2 - 2$:



b Whitehill, WPI

Exercises (homework)

- What about $x^4 + xy + x^2$?
    - Its Hessian is: $\mathbf{H} = \left[\begin{array}{cc} 12x^2 + 2 & 1 \\ 1 & 0 \end{array} \right]$
    - You either have to prove that **H** is always PSD, or find a counter-example (homework 2).
- It can be shown (homework 2) that the MSE loss of a 2-layer linear neural network is convex.

Convex ML models

- Prominent convex models in ML include linear regression, logistic regression, softmax regression, and support vector machines (SVM).

- However, models in deep learning are generally not convex.

  - Much DL research is devoted to how to optimize the weights to deliver good generalization performance.