Chapter 5

Model selection and overfitting

Neural networks and deep learning

Model Selection

In neural networks, there exist several free parameters: learning rate, batch size, no of layers, number of neurons, etc.

Every set of parameters of the network leads to a specific model.

How do we determine the “optimum” parameter(s) or the model for a given regression or classification problem?

Selecting the best model with the best parameter values

Performance estimates

How do we measure the performance of the network?

Some metrics:

1. Mean-square error/Root-mean square error for **regression** — the mean-squared error or its square root. A measure of the deviation from actual.

$$MSE = \frac{1}{P}\sum_{p=1}^{P}\sum_{k=1}^{K}\left(d_{pk} - y_{pk}\right)^2 \text{ and } RMSE = \sqrt{\frac{1}{P}\sum_{p=1}^{P}\sum_{k=1}^{K}\left(d_{pk} - y_{pk}\right)^2}$$

2. Classification error of a **classifier**.

$$classification\ error = \sum_{p=1}^{P} 1\left(d_p \neq y_p\right)$$

where d_p is the target and y_p is the predicted output of pattern p. $1(\cdot)$ is the indicator function.

True error or apparent error?

Apparent error (training error): the error on the training data. What the learning algorithm tries to optimize.

True error: the error that will be obtained in use (i.e., over the whole *sample space*). What we want to optimize *but* unknown.

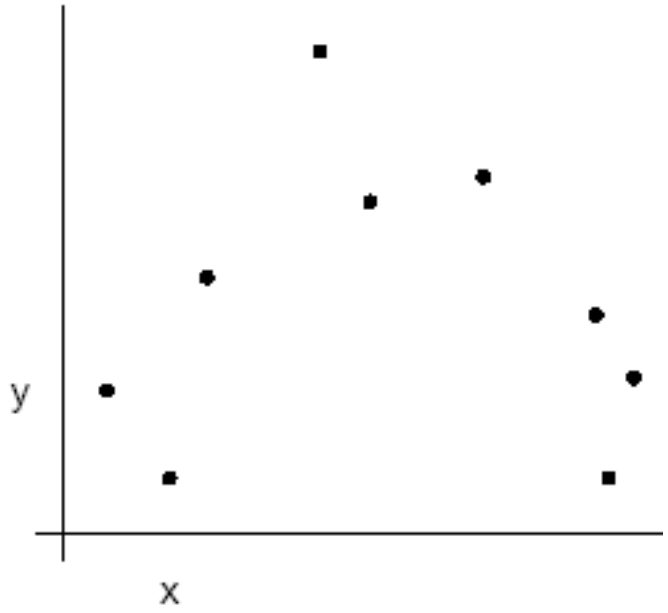
However, the **apparent error** is not always a good estimate of the **true error**. It is just an optimistic measure of it.

Test error: (out-of-sample error) an estimate of the true error obtained by testing the network on some independent data.

Generally, a larger test set helps provide a greater confidence on the accuracy of the estimate.

Example: Regression

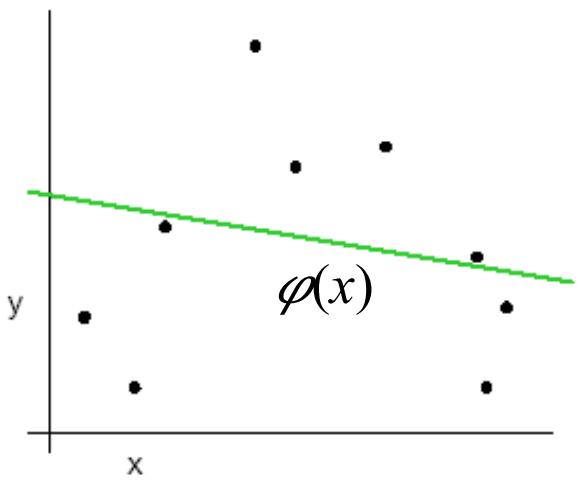
Given the following sample data:



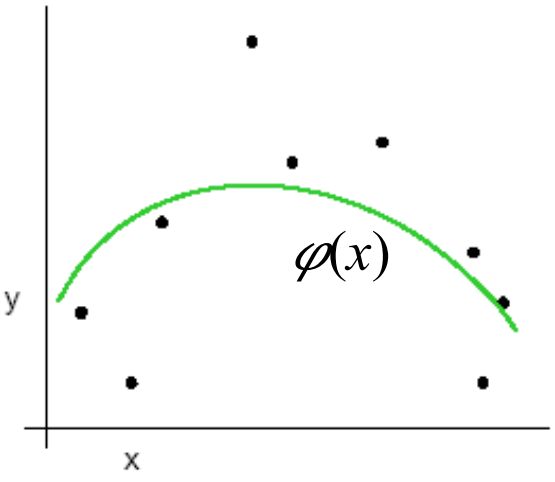
Approximate $y \approx \varphi(x)$ using an NN

Example: Regression

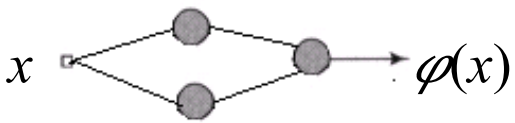
Which one is the best approximation model?

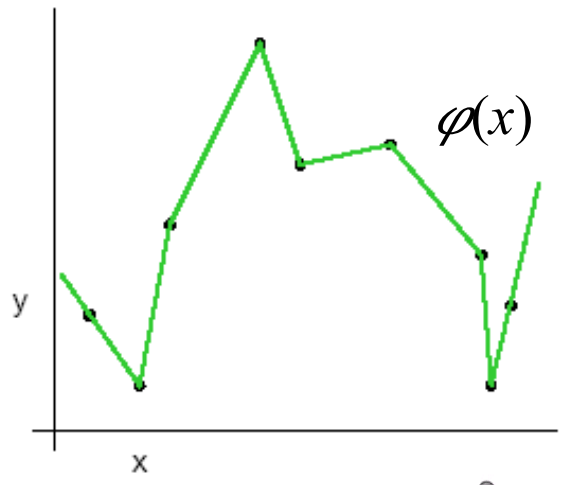


Linear regression, Linear Activation fn

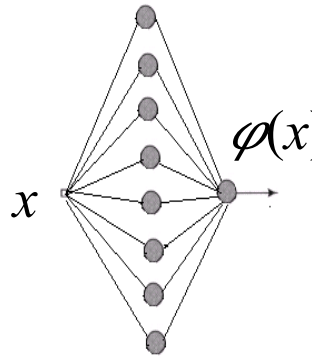


Quadratic (non-linear) regression, Sigmoid Activation fn



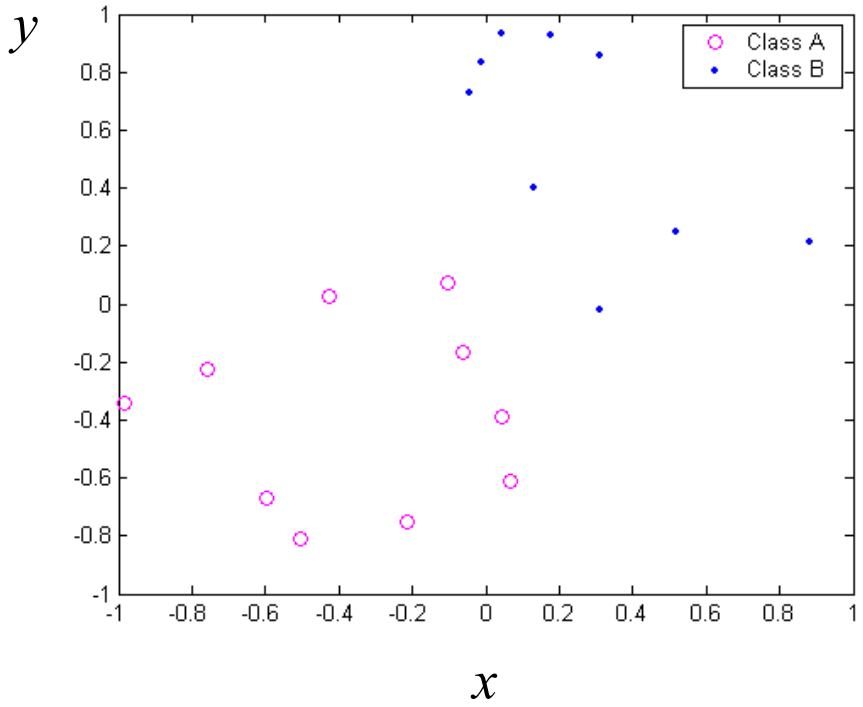


Piecewise Linear regression, Linear Activation fn



Example: Classification

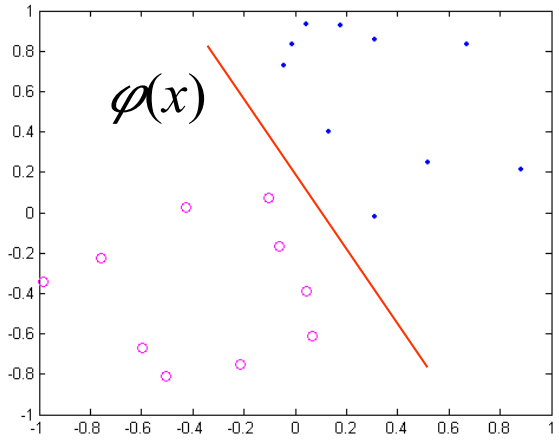
Given the following sample data:



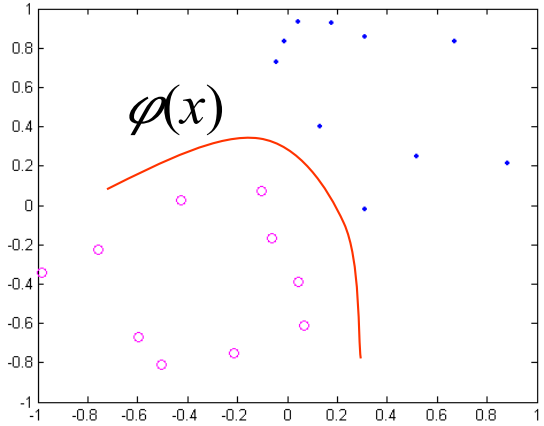
Classify the following data using an NN.

Example: Classification

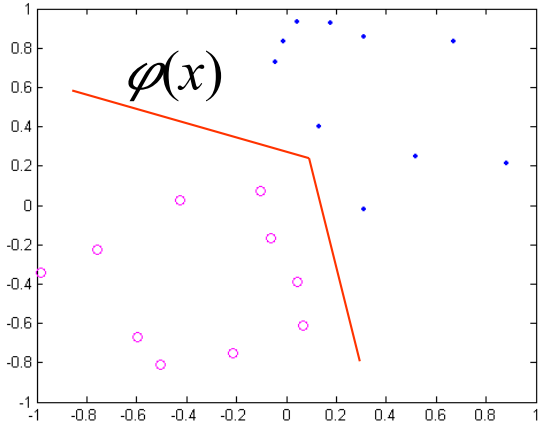
Which one is the best classification model?



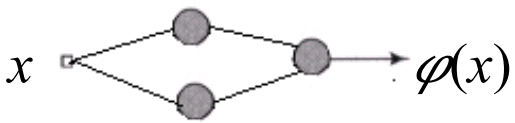
Linear Classification, Threshold Activation fn

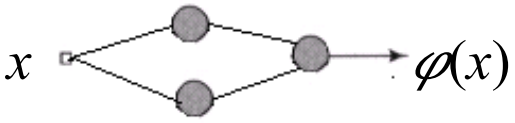


Quadratic Classification, *Sigmoid Activation fn*



Piecewise Linear Classification, *Linear Activation fn*





Estimation of true error

***Intuition**:* Choose the model with the best fit to the data?

***Meaning**:* Choose the model that provides the lowest error rate on the entire sample population. Of course, that error rate is the *true error rate*.

"However, to choose a model, we must first know how to **estimate the error** of a model."

The entire **sample population** is often unavailable and only **example data** is available.

Validation

In real applications, we only have access to a finite set of examples, usually smaller than we wanted.

Validation is the approach to use the entire example data available to build the model and estimate the error rate. The validation uses a part of the data to select the model, which is known as the *validation set.*

Validation attempts to solve fundamental problems encountered:

- The model tends to *overfit* the training data. It is not uncommon to have 100% correct classification on training data.
- There is no way of knowing how well the model performs on *unseen data*
- The *error rate estimate* will be overly optimistic (usually lower than the true error rate) . Need to get an unbiased estimate.

Validation Methods

An effective approach is to split the entire data into subsets, i.e., Training/Validation/Testing datasets.

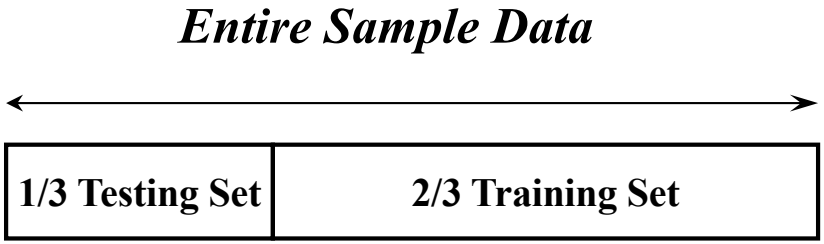
Some Validation Methods:

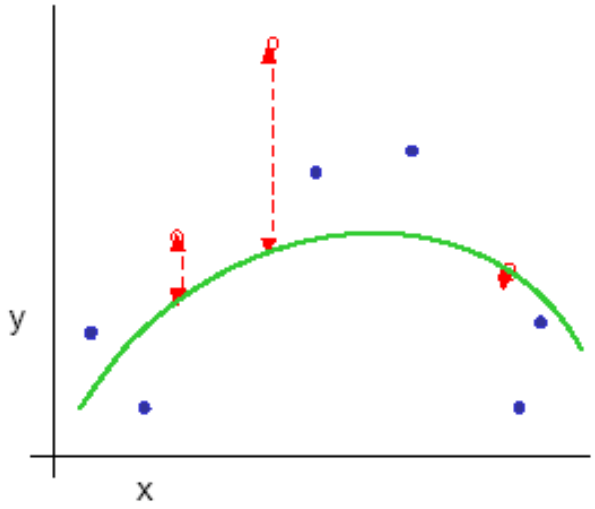
- The Holdout (1/3 - 2/3 rule for test and train partitions)
- Re-sampling techniques
    - Random Subsampling
    - K-fold Cross-Validation
    - Leave one out Cross-Validation
- Three-way data splits (train-validation-test partitions)

Holdout Method

Split entire dataset into two sets:

- **Training set** (blue): used to train the classifier
- **Testing set** (red): used to estimate the error rate of the trained classifier on unseen data samples

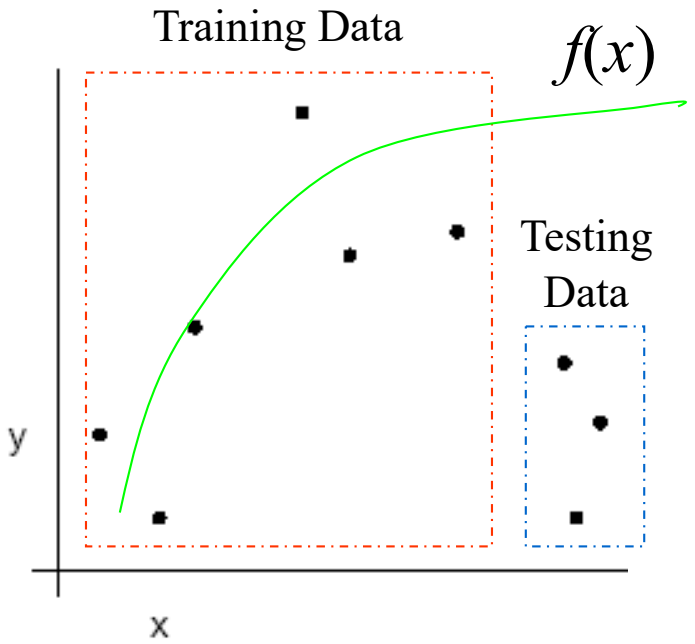




Holdout Method

The holdout method has two basic drawbacks:

- By setting some samples for testing, the training dataset becomes smaller
- Use of a single train-and-test experiment, could lead to misleading estimate if an "unfortunate" split happens



An "unfortunate" split: training data may not cover the space of testing data

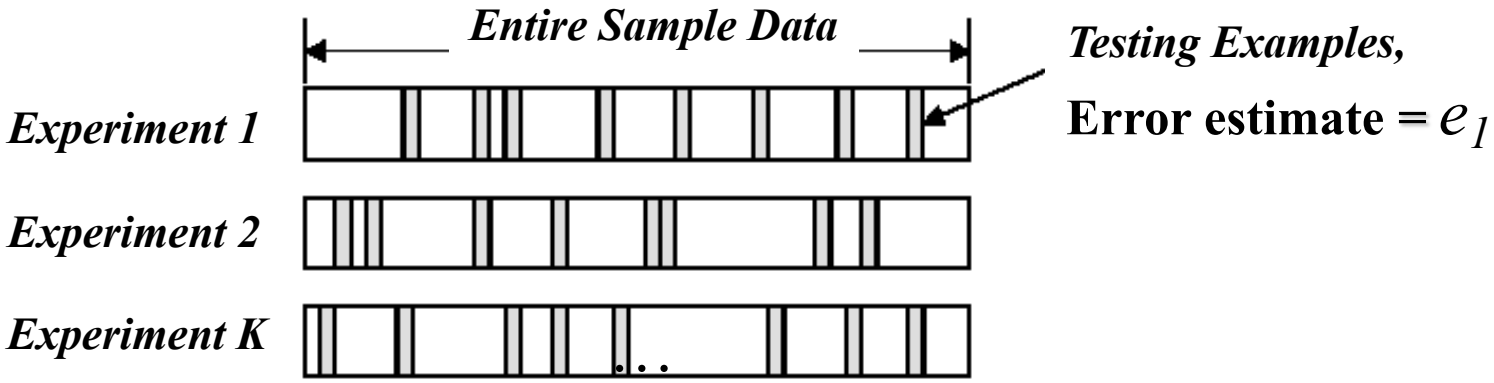
Random Sampling Methods

Limitations of the holdout can be overcome with a family of resampling methods at the expense of more computations:

- Random Subsampling
- K-Fold Cross-Validation
- Leave-one-out (LOO) Cross-Validation

K Data Splits Random SubSampling

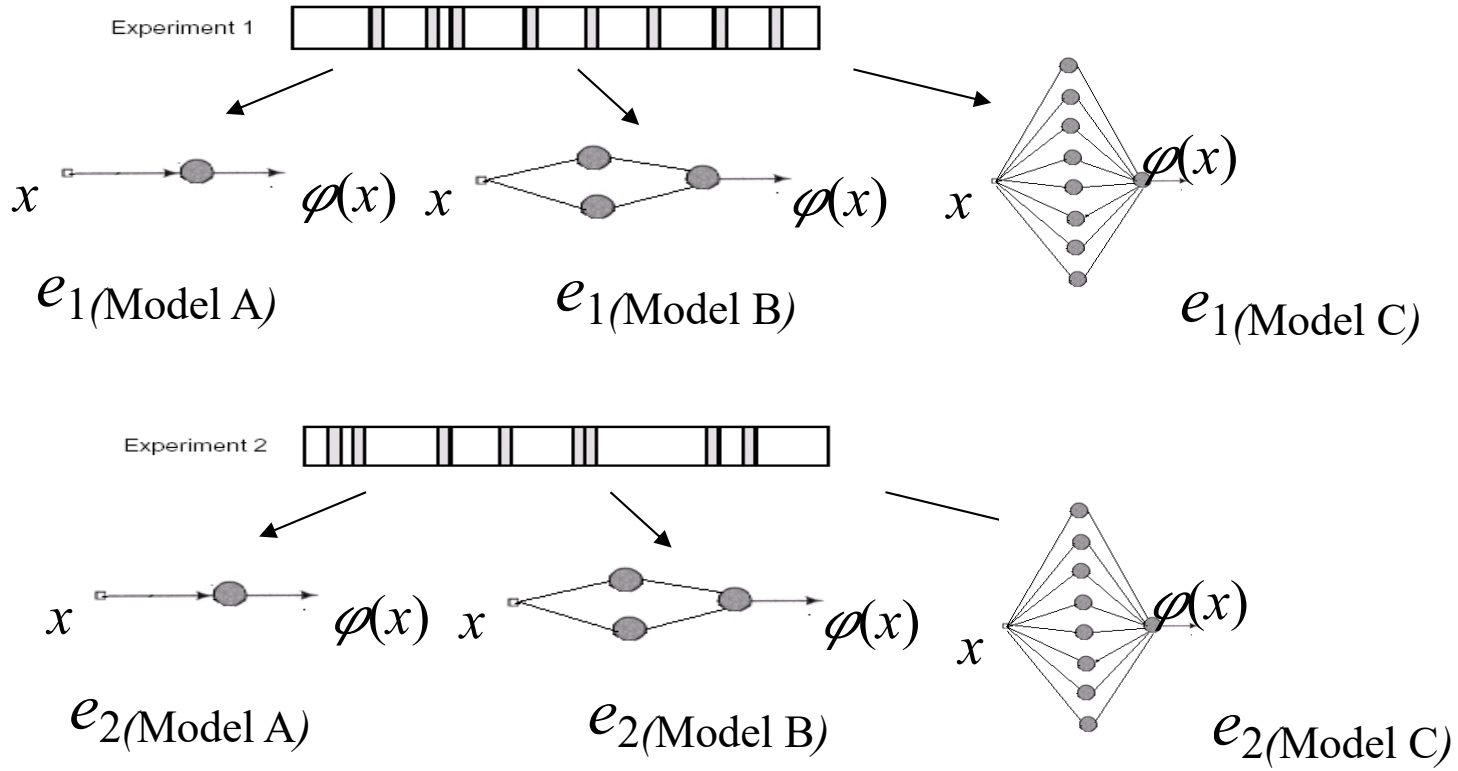
Random Subsampling performs *K* data splits of the dataset for training and testing.



Each split randomly selects a (fixed) no. of examples. For each data split we retrain the classifier from scratch with the training data. Let the error estimate obtained for ith split (experiments) be e_i.

$$Average\ test\ error = \frac{1}{K}\sum_{i=1}^{K} e_i$$

Example: K=2 Data Splits Random SubSampling



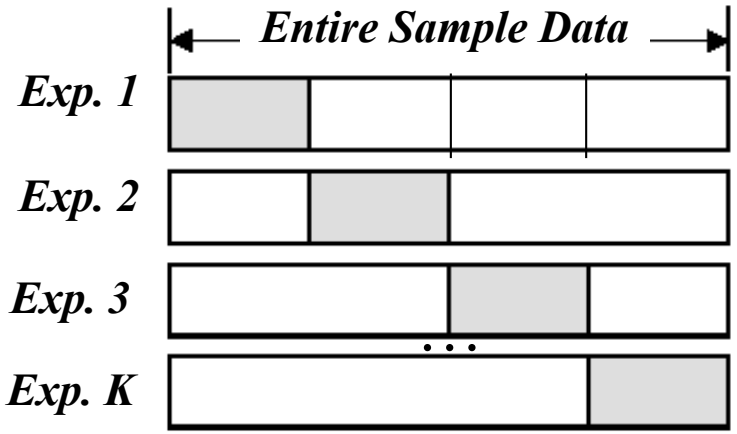
$$Test\ error_{(M)} = \frac{1}{2}(e_{1(M)} + e_{2(M)})$$

Choose the model with the best test error, i.e., lowest average test error!

K-fold Cross-Validation

Create a *K*-fold partition of the the dataset:

• For each of *K* experiments, use *K*-1 folds for training and the remaining one-fold for testing

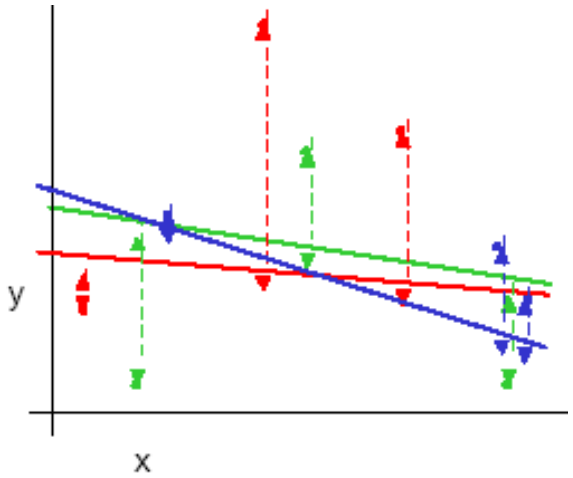


Let the error estimate for *i*th experiment on test partition be e_i.

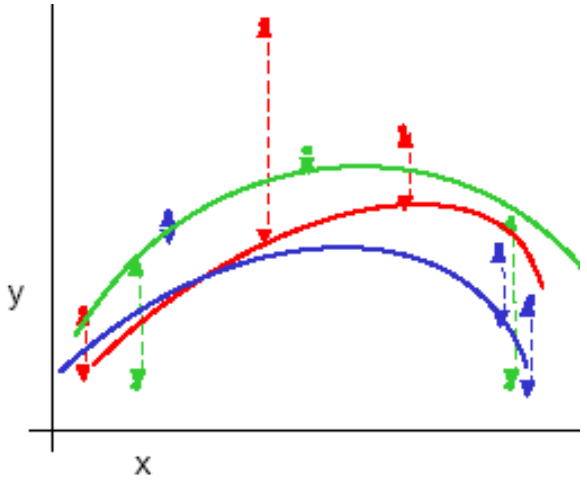
Cross-validation error $\frac{1}{K}\sum_{i=1}^{K} e_i$

K-fold cross validation is similar to Random Subsampling. The *advantage* of *K*-Fold Cross validation is that all examples in the dataset are eventually used for both training and testing

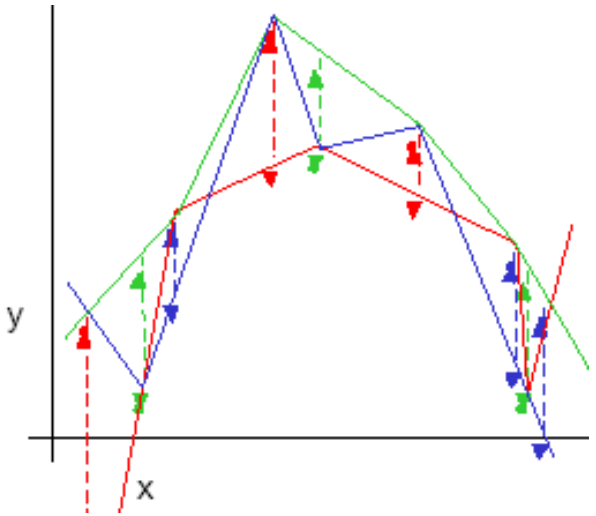
Example: 3-fold Cross-Validation



e_1(Model A)



e_1(Model B)

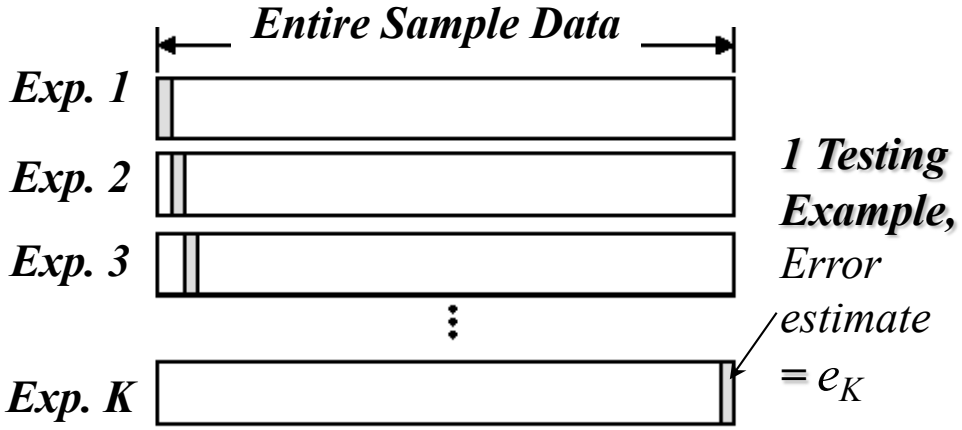


e_1(Model C)

Leave-One-Out (LOO) Cross-Validation

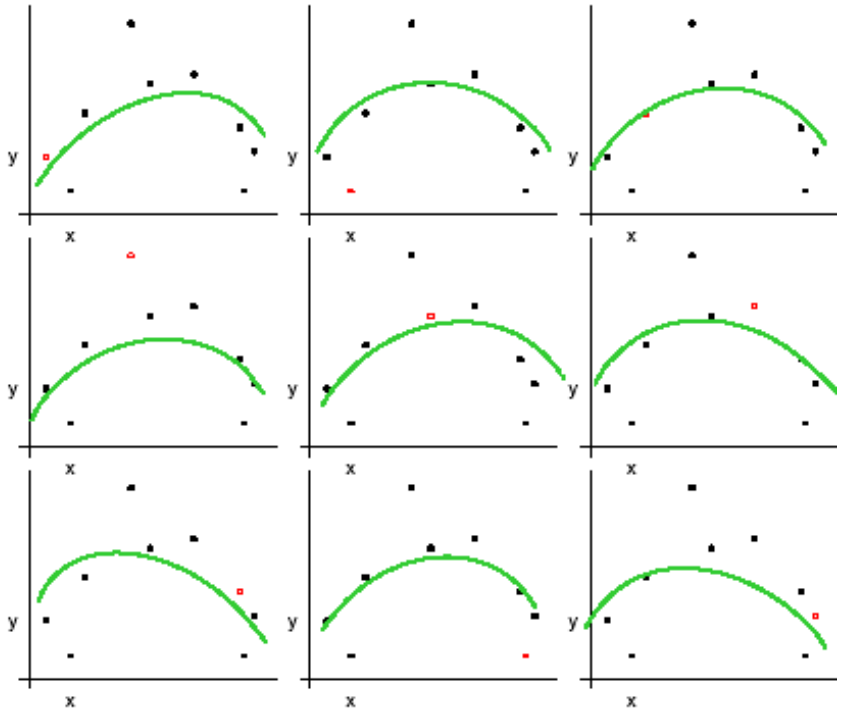
Leave-One-Out is the degenerate case of *K*-Fold Cross Validation, where *K* is chosen as the total number of examples:

- For a dataset with N examples, perform N experiments, i.e., $N = K$.
- For each experiment use N-1 examples for training and the remaining one example for testing.

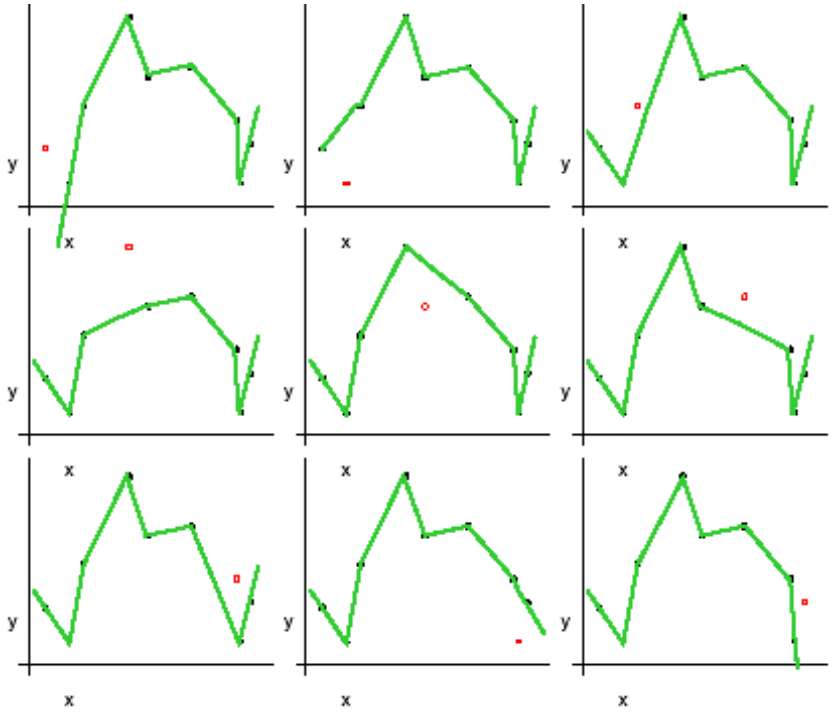


LOO CV error $\square \frac{1}{K} \sum_{i \square 1}^{K} e_i$

Example: Leave-One-Out Cross-Validation



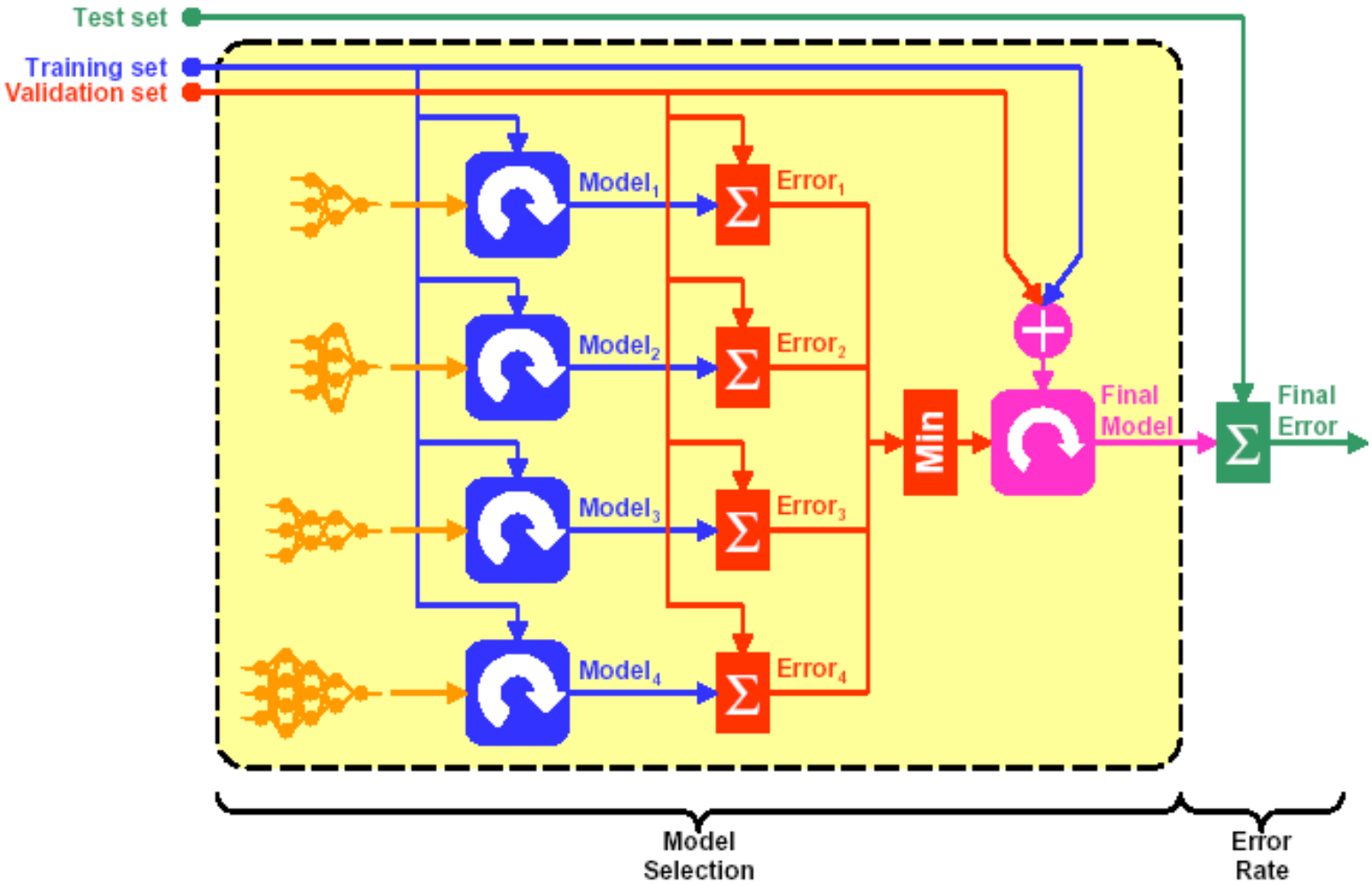
LOOCV Quadratic
Approximation



LOOCV Piecewise
Linear Approximation

Three-Way Data Splits Method

Dataset is partitioned in to training set, **validation set**, and testing set.



Three-Way Data Splits Method

If model selection and true error estimates are to be computed simultaneously, the data needs to be divided into three disjoint sets:

• **Training set**: examples for *learning* to fit the parameters of several possible classifiers. In the case of DNN, we would use the training set to find the "optimal" weights with the gradient descent rule.

• **Validation set**: examples to *determine* the error J_m of different models m, using the validation set. The optimal model m^* is given by

$$m^* = \underset{m}{\arg min}\, J_m$$

• **Training + Validation set:** combine examples used to re-train/redesign $model_{m^*,}$ and find new "optimal" weights and biases.

• **Test set**: examples used only to *assess* the performance of a *trained model* m^*. We will use the test data to estimate the error rate after we have trained the final model with train + validation data.

Three-Way Data Splits Method

Why separate test and validation sets?

• The error rate estimate of the final model on validation data will be biased (smaller than the true error rate) since the validation set is also used to select in the process of final model selection.

• After assessing the final model, an independent test set is required to estimate the performance of the final model.

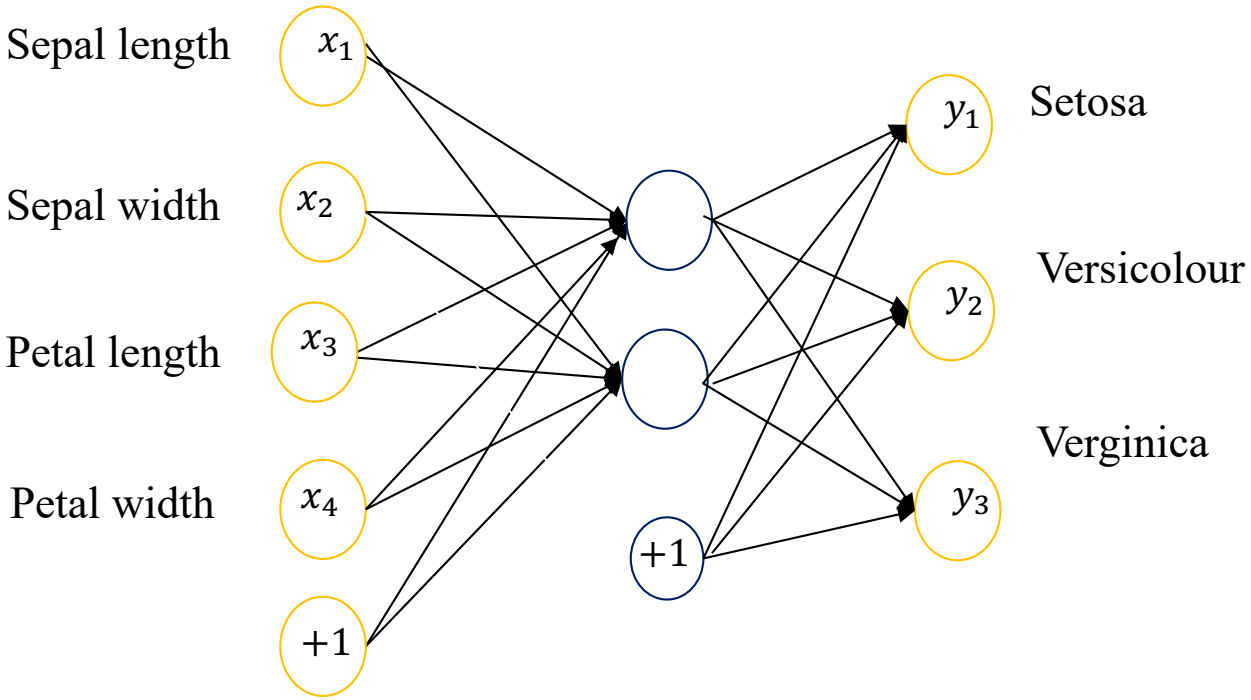
"NO FURTHERING TUNNING OF THE MODEL IS ALLOWED!"

Examples 1-3: Iris dataset

https://archive.ics.uci.edu/ml/datasets/Iris

Three classes of iris flower: Setosa, versicolour, and virginica
Four features: Sepal length, sepal width, petal length, petal width
150 data points

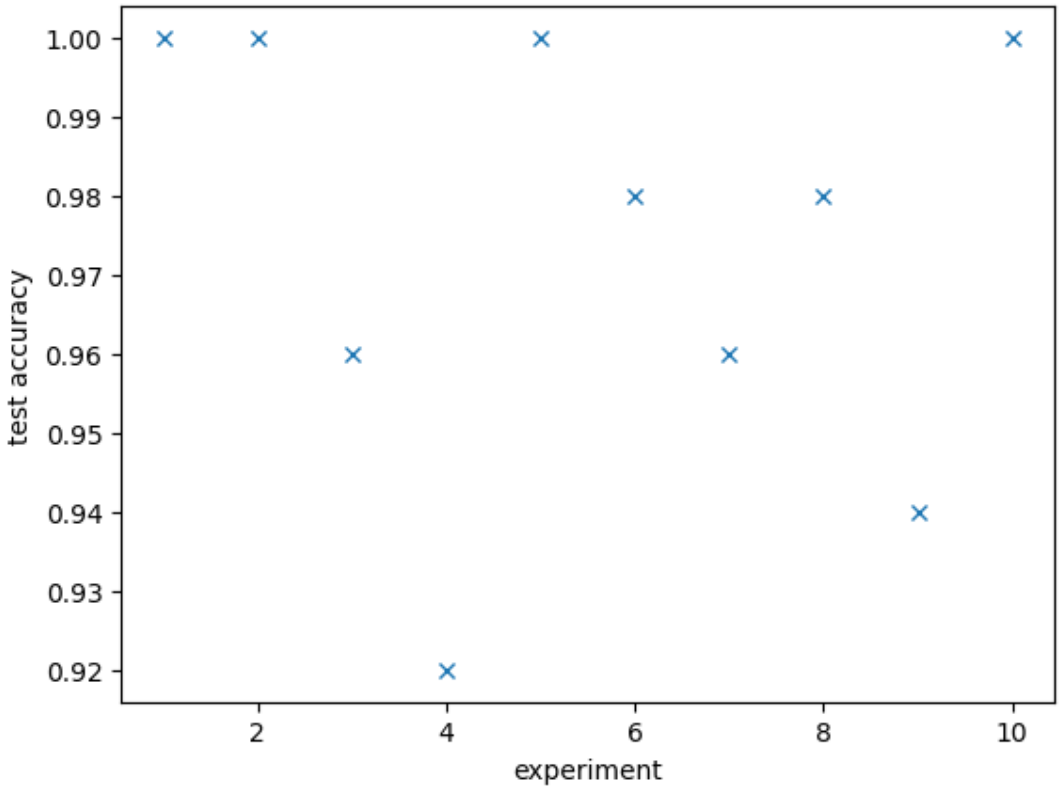
DNN with one hidden layer. **Determine number of hidden neurons?**



Example 1a: Random subsampling

150 data points
In each experiment, 50 points for testing and 100 for training

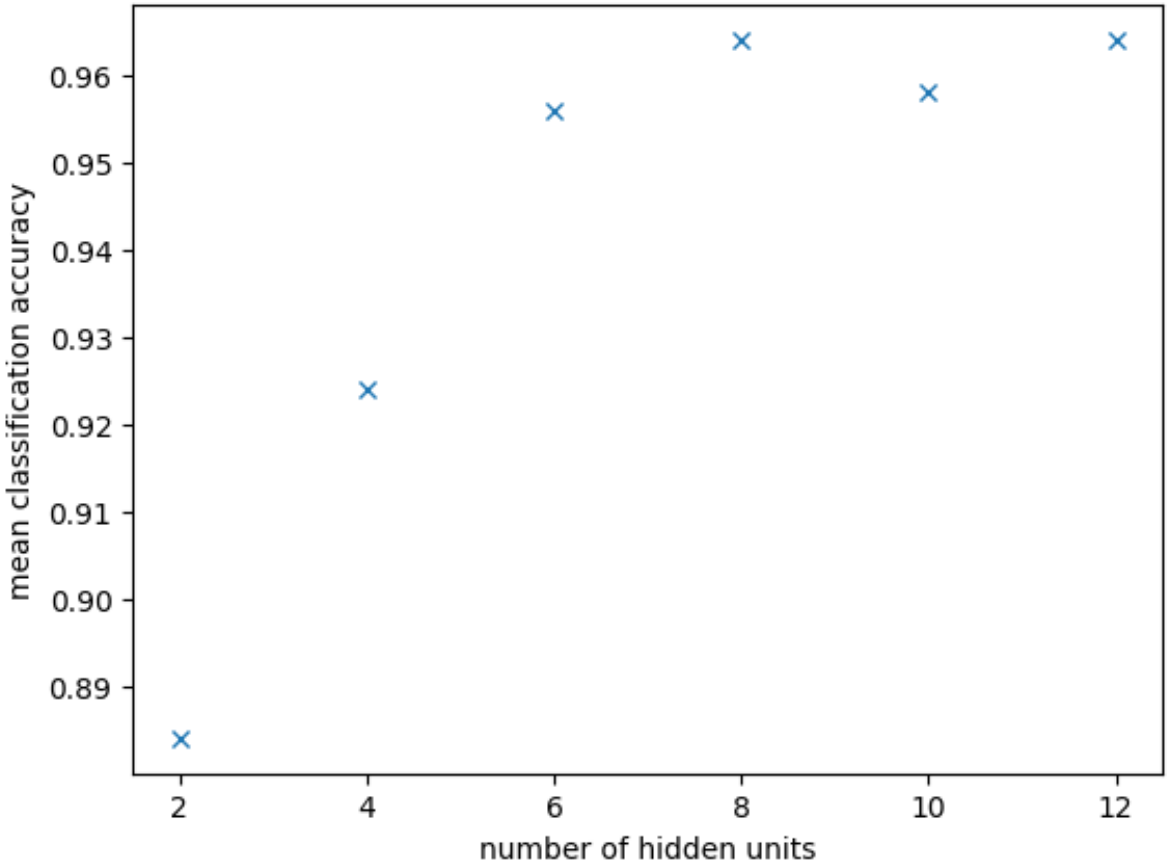
Example: 5 hidden neurons, 10 experiments



Mean accuracy = 97.4%

Example 1b

For different number of hidden units, misclassification errors in 10 experiments



Optimum number of hidden units = 8

Accuracy = 96.4%

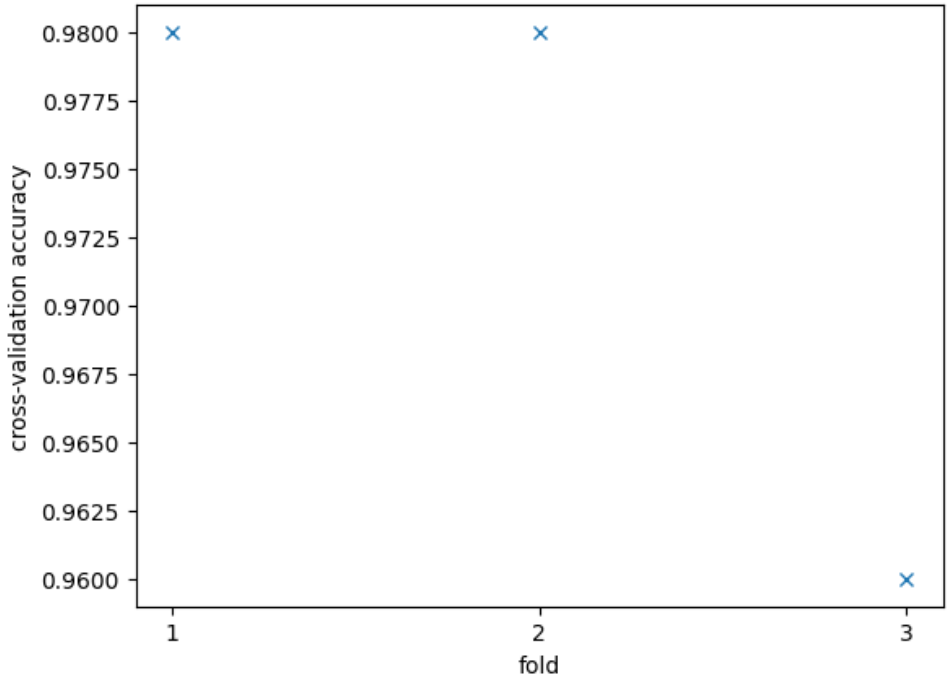
Example 2a: Cross validation

150 data points

3-fold cross validation: 50 data points in one fold.

Two folds are used for training and the remaining fold for testing

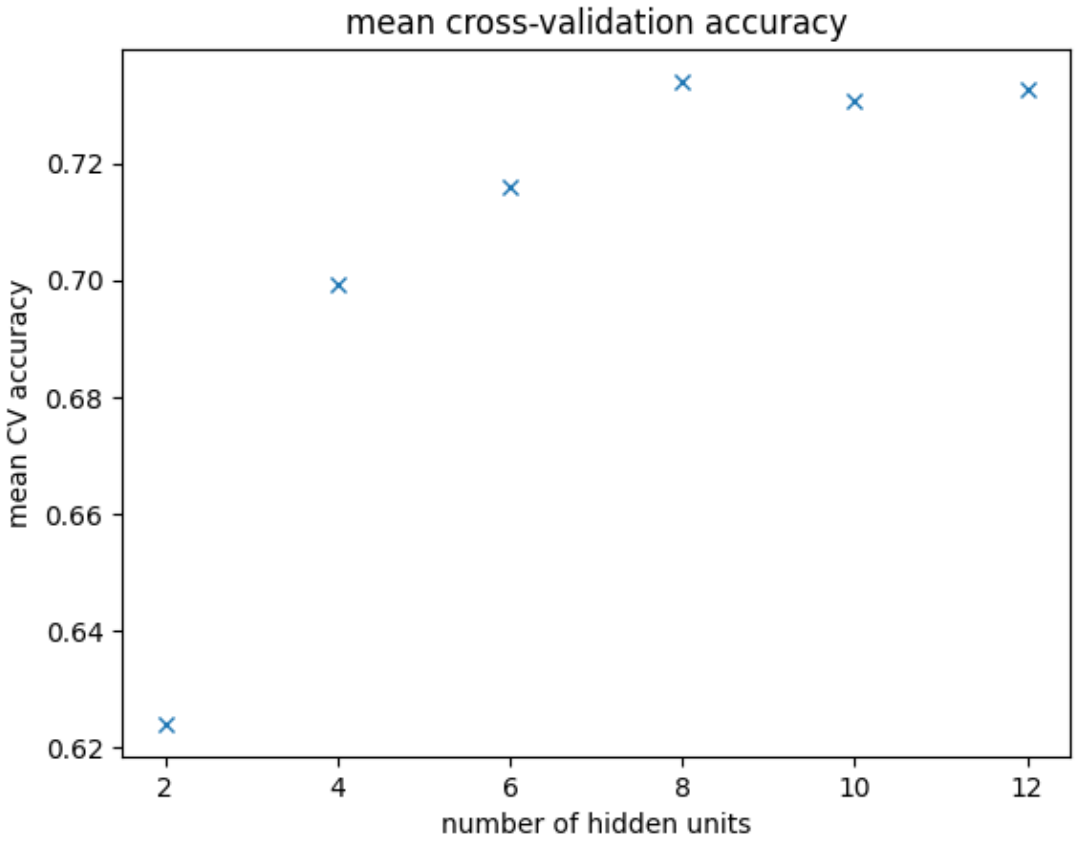
Example: hidden number of units = 5



3-fold cross-validation (CV) accuracy = 97.3%

Example 2b

Mean CV error for 10 experiments for different number of hidden units:

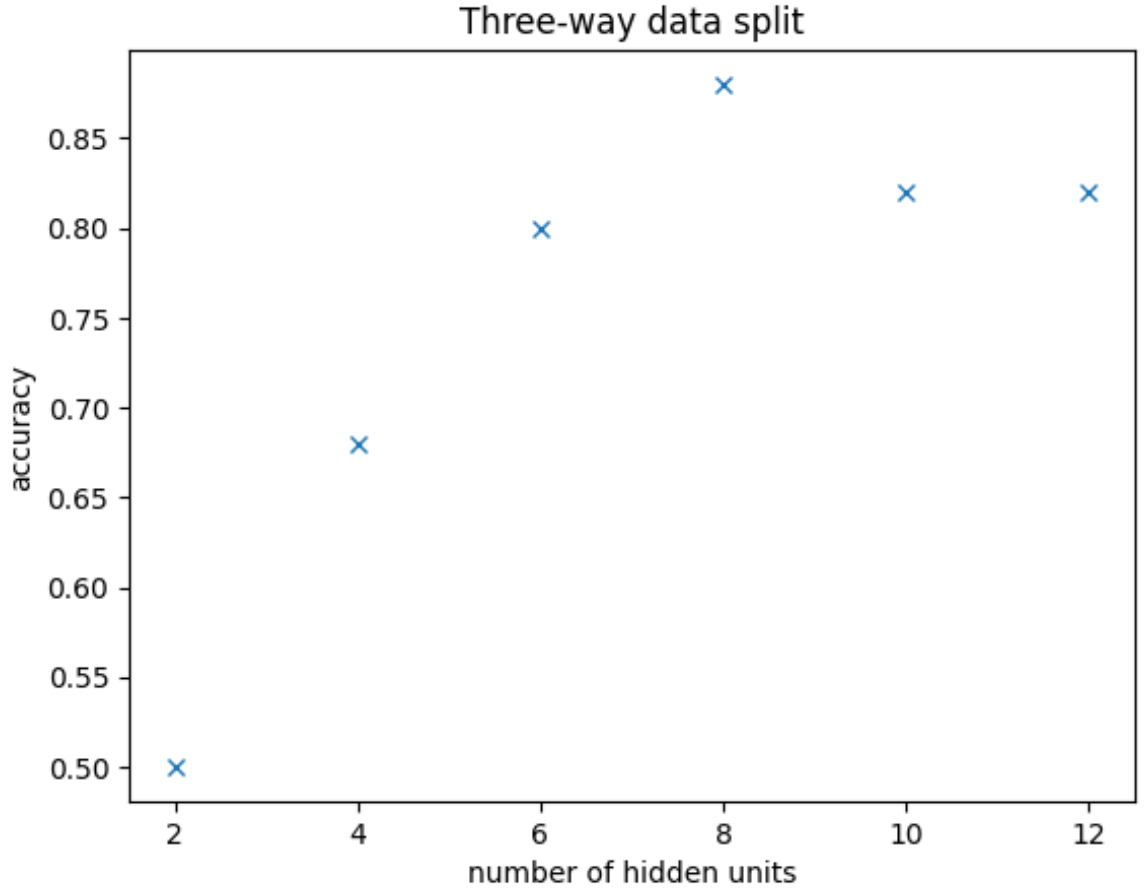


Optimum number of hidden neurons = 8
Cross-validation accuracy = 73.4%

Example 3a: Three-way data splits

150 data points

50 data points each for training, for validation, and for testing.

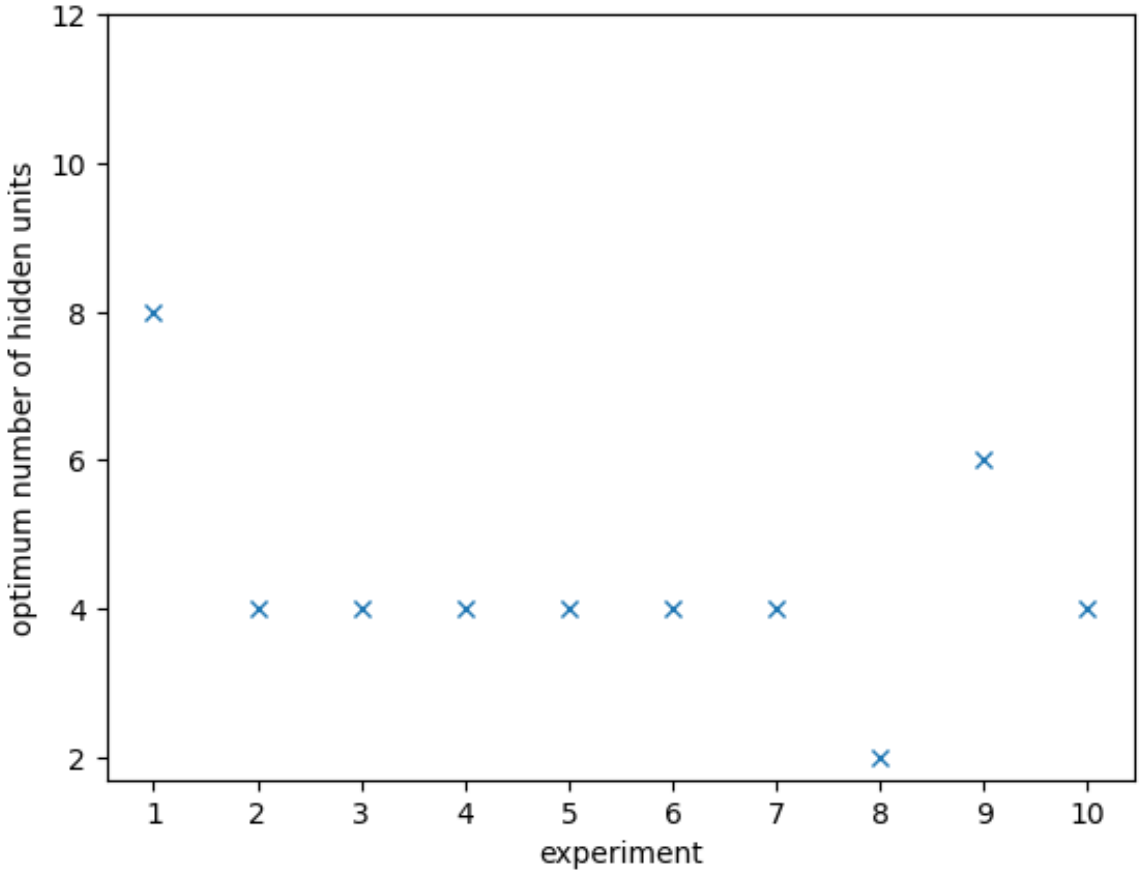


Optimum number of hidden neurons = 8

Accuracy = 88.0%

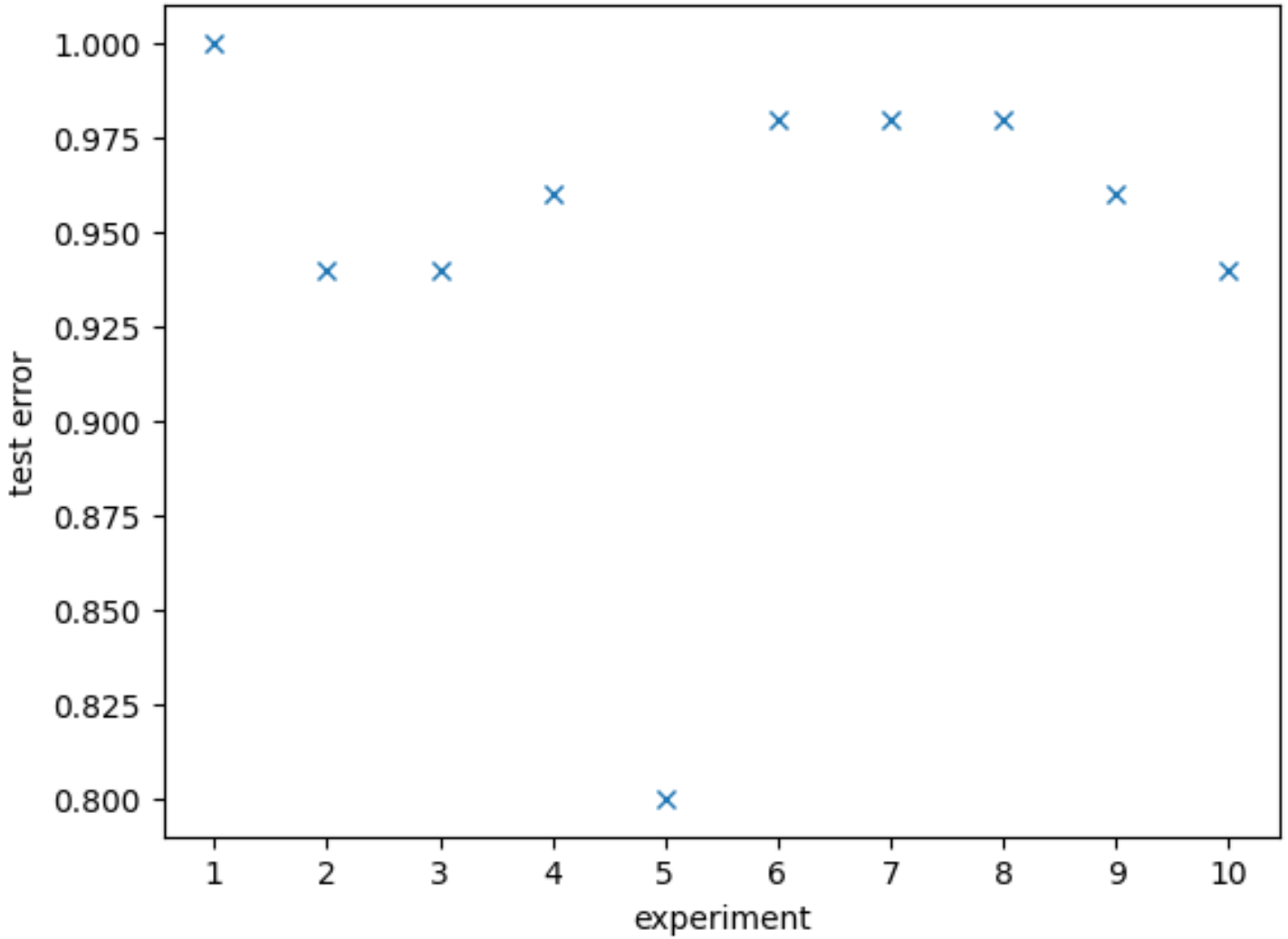
Example 3b

One way to further improve the results is to repeat it for many experiments, say for 10 experiments:



Optimum number of hidden neurons = 4

Example 3b



Test accuracy = 93.4% (average)

Model Complexity

Complex models: models with many adjustable weights and biases will

- more likely to be able to solve the required task,
- more likely to memorize the training data without solving the task.

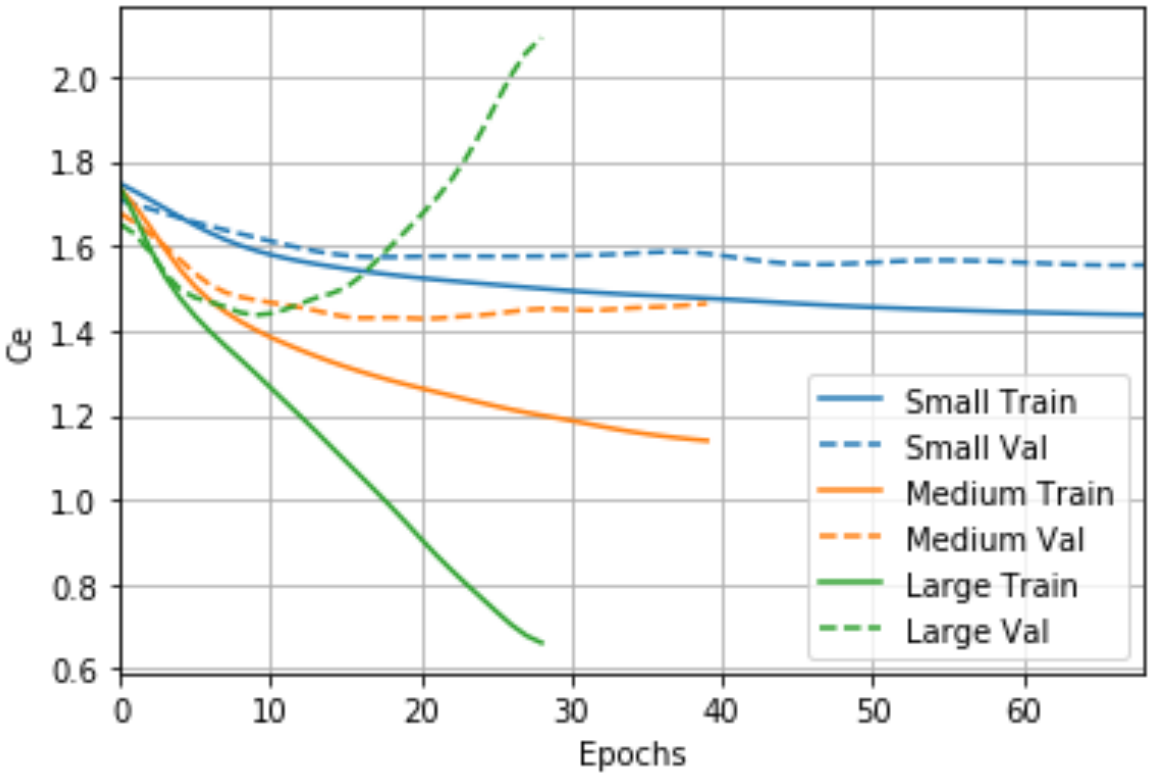
Simple models: The simpler the model that can learn from the training data is more likely to generalize over the entire sample space. May not learn the problem well.

This is the fundamental trade-off:

• Too simple — cannot do the task because not enough parameters to learn. e.g., 5 hidden neurons. (*underfitting*)

• Too complex — cannot generalize from small and noisy datasets well, e.g., 20 hidden neurons. (*overfitting*)

Overfitting and underfitting



Small model is **underfitting**
Medium model seems just right
Large model is **overfitting**

Overfitting

Overfitting is one of the problems that occur during training of neural networks, which drives the training error of the network to a very small value at the expense of the test error. The network learns to respond correctly to the training inputs by remembering them too much but is unable to generalize to produce correct outputs to novel inputs.

- Overfitting happens when the amount of training data is inadequate in comparison to the number of network parameters to learn.

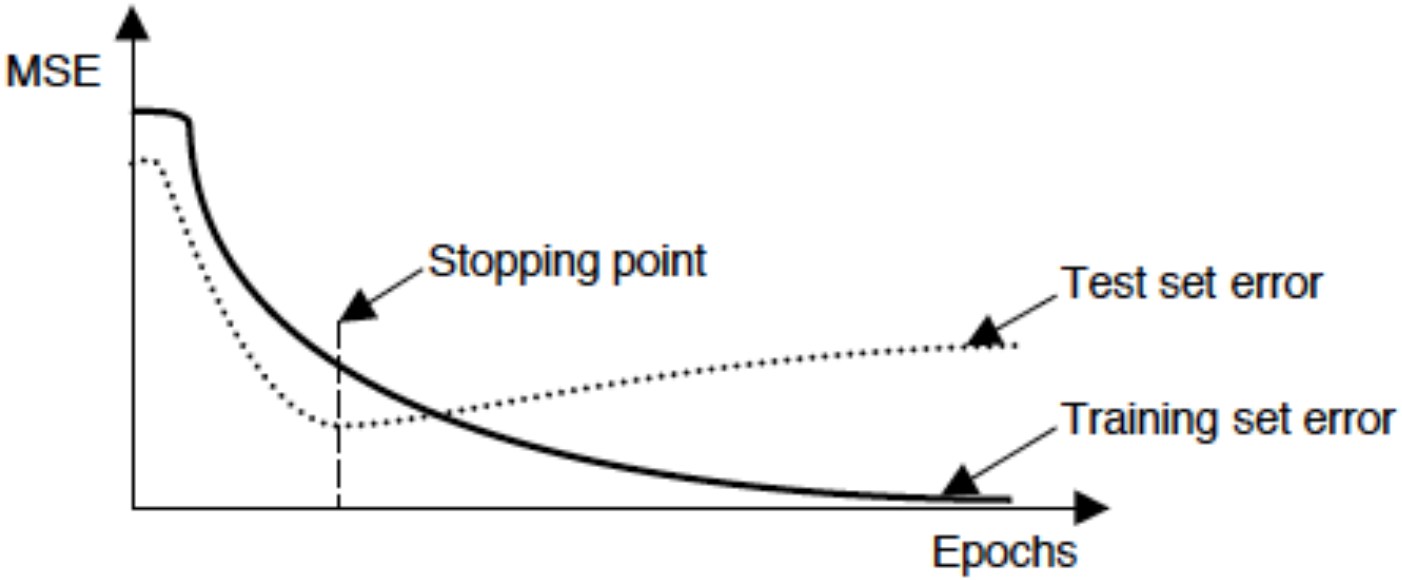
- Overfitting occurs when the weights and biases become too large and are fine-tuned to remember the training patterns too much.

- Even your model is right, too much training can cause overfitting.

Methods to overcome overfitting

1. Early stopping
2. Regularization of weights
3. Dropouts

Early stopping



Training of the network is to be stopped when the validation error starts increasing. Early stopping can be used in test/validation by stopping when the validation error is minimum.

Regularization of weights

During overfitting, some weights attain large values to reduce training error, jeopardizing its ability to generalizing. In order to avoid this, a *penalty* term (*regularization* term) is added to the cost function.

For a network with weights $\boldsymbol{W} = \{w_{ij}\}$ and bias $\boldsymbol{b}$, the penalized (or regularized) cost function $J_1(\boldsymbol{W}, \boldsymbol{b})$ is defined as

$$J_1 = J + \beta_1 \sum_{i,j} |w_{ij}| + \beta_2 \sum_{ij} (w_{ij})^2$$

where $J(\boldsymbol{W}, \boldsymbol{b})$ is the standard cost function (i.e., m.s.e. or cross-entropy),

$$L^1 - norm = \sum_{i,j} |w_{ij}|$$

$$L^2 - norm = \sum_{ij} (w_{ij})^2$$

And β_1 and β_2 are known as L^1 and L^2 regularization (penalty) constants, respectively. These penalties discourage weights from attaining large values.

L2 regularization of weights

Regularization is usually not applied on bias terms. L^2 regularization is most popular on weights.

$$J_1 = J + \beta_2 \sum_{ij} \left(w_{ij}\right)^2$$

Gradient of the regularized cost wrt weights:

$$\nabla_{\boldsymbol{W}} J_1 = \nabla_{\boldsymbol{W}} J + \beta_2 \frac{\partial \left(\sum_{ij} {w_{ij}}^2\right)}{\partial \boldsymbol{W}} \qquad \text{(A)}$$

L2 regularization of weights

$$L^2 \text{ norm} = \sum_{ij} \left(w_{ij}\right)^2 = {w_{11}}^2 + {w_{12}}^2 + \cdots {w_{Kn}}^2$$

$$\frac{\partial \left(\sum_{ij} \left(w_{ij}\right)^2\right)}{\partial \boldsymbol{W}} = \begin{pmatrix} \frac{\partial \sum \left(w_{ij}\right)^2}{\partial w_{11}} & \frac{\partial \sum \left(w_{ij}\right)^2}{\partial w_{12}} & \cdots & \frac{\partial \sum \left(w_{ij}\right)^2}{\partial w_{1k}} \\ \frac{\partial \sum \left(w_{ij}\right)^2}{\partial w_{21}} & \frac{\partial \sum \left(w_{ij}\right)^2}{\partial w_{22}} & \cdots & \frac{\partial \sum \left(w_{ij}\right)^2}{\partial w_{2k}} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial \sum \left(w_{ij}\right)^2}{\partial w_{n1}} & \frac{\partial \sum \left(w_{ij}\right)^2}{\partial w_{n2}} & \cdots & \frac{\partial \sum \left(w_{ij}\right)^2}{\partial w_{nK}} \end{pmatrix} = 2\boldsymbol{W}$$

L2 regularization of weights

Substituting in (A):

$$\nabla_{\boldsymbol{W}} J_1 = \nabla_{\boldsymbol{W}} J + \beta_2 \frac{\partial\left(\sum_{ij} {w_{ij}}^2\right)}{\partial \boldsymbol{W}} = \nabla_{\boldsymbol{W}} J + 2\beta_2 \boldsymbol{W}$$

For gradient decent learning that uses regularized cost function:

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \nabla_{\boldsymbol{W}} J_1$$

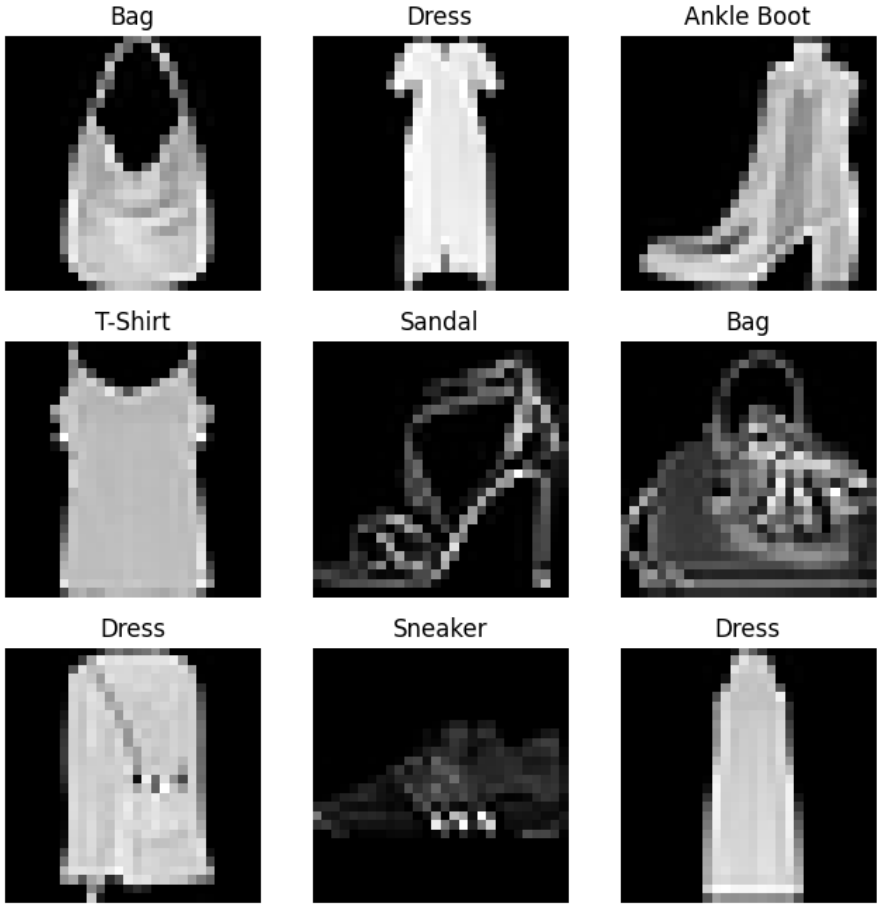
Substituting above:

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha(\nabla_{\boldsymbol{W}} J + \beta \boldsymbol{W})$$

where $\beta = 2\beta_2$
β is known as the *weight decay parameter*.

That is for L^2 regularization, the weight matrix is weighted by decay parameter and added to the gradient term.

Fashion MNIST dataset



https://github.com/zalandoresearch/fashion-mnist

Example 4: Early stopping and weight decay

DNN with [784, 400, 400, 400, 10] architecture.
Let's implement L2 regularization with $\beta = 0.0001$
Use early stopping to terminate learning.

```
class NeuralNetwork(nn.Module):
    def __init__(self, hidden_size=500):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(32*32*3, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, 10),
            nn.Softmax(dim=1)
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```

Example 4

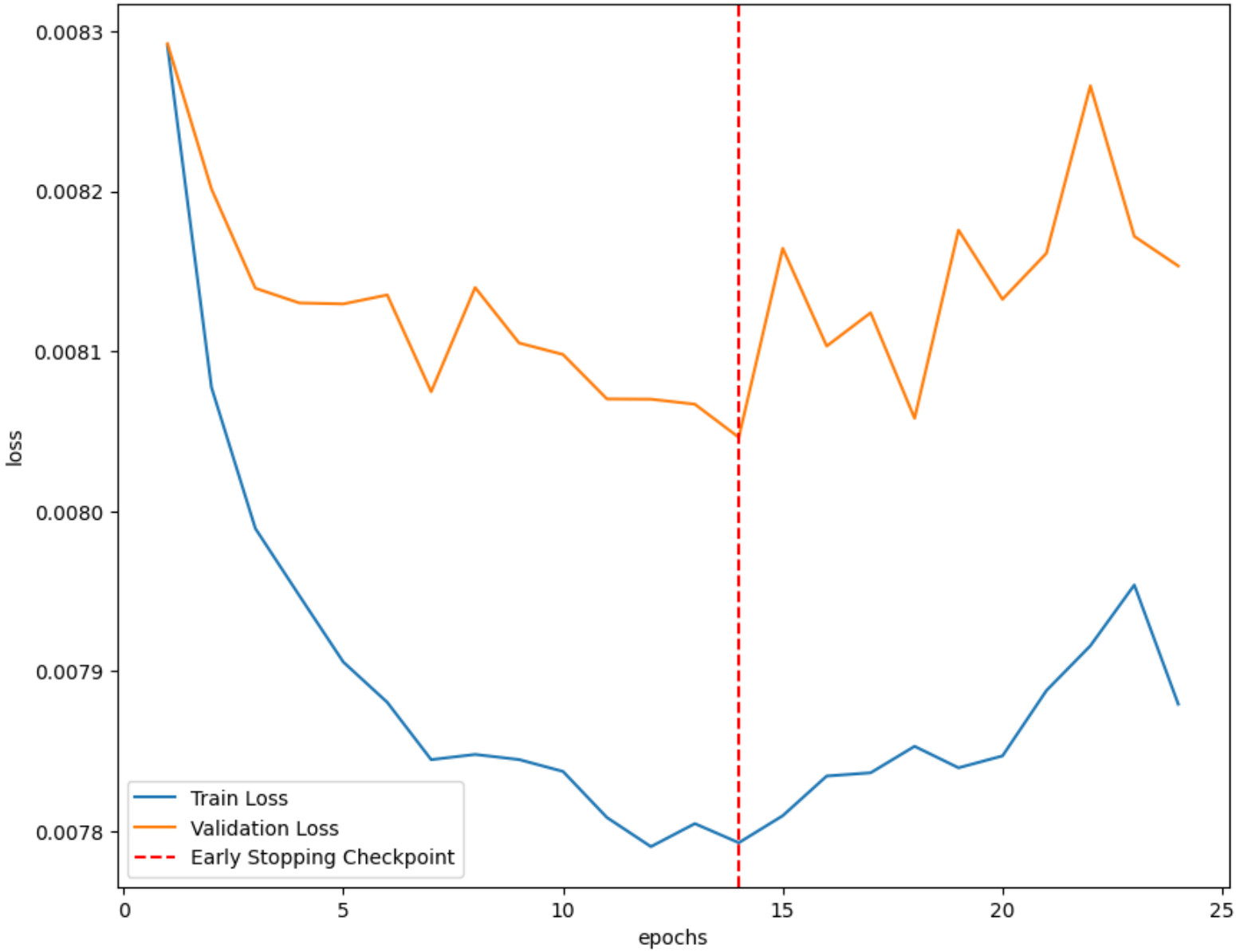
```
class EarlyStopper:
    def __init__(self, patience=5, min_delta=0):
        self.patience = patience
        self.min_delta = min_delta
        self.counter = 0
        self.min_validation_loss = np.inf

    def early_stop(self, validation_loss):
        if validation_loss < self.min_validation_loss:
            self.min_validation_loss = validation_loss
            self.counter = 0
        elif validation_loss > (self.min_validation_loss + self.min_delta):
            self.counter += 1
            if self.counter >= self.patience:
                return True
        return False
```

Example 4

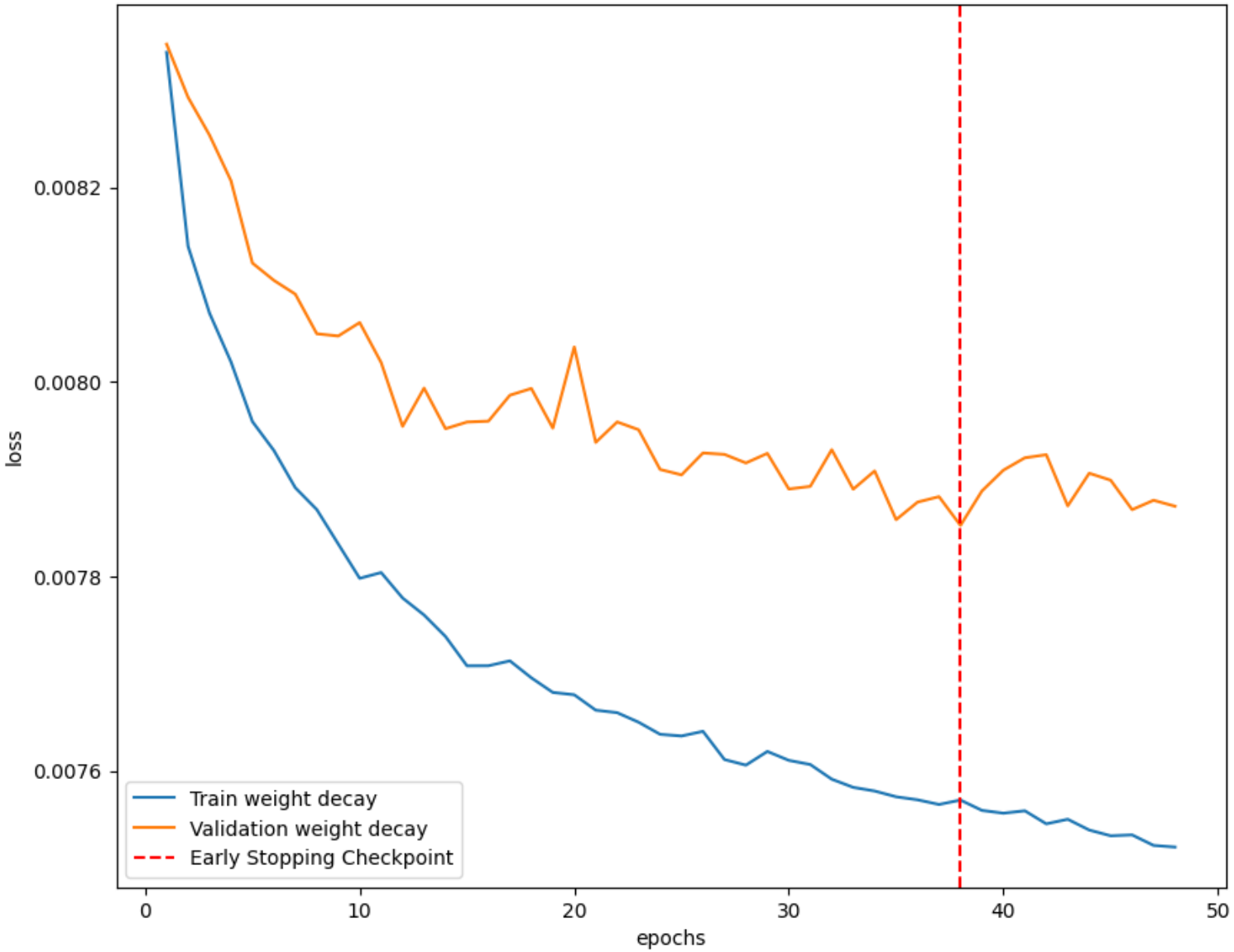
```
model = NeuralNetwork()
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate, weight_decay=0.001)
early_stopper = EarlyStopper(patience=patience, min_delta)
```

```
if early_stopper.early_stop(test_loss):
        print("Done!")
        break
```

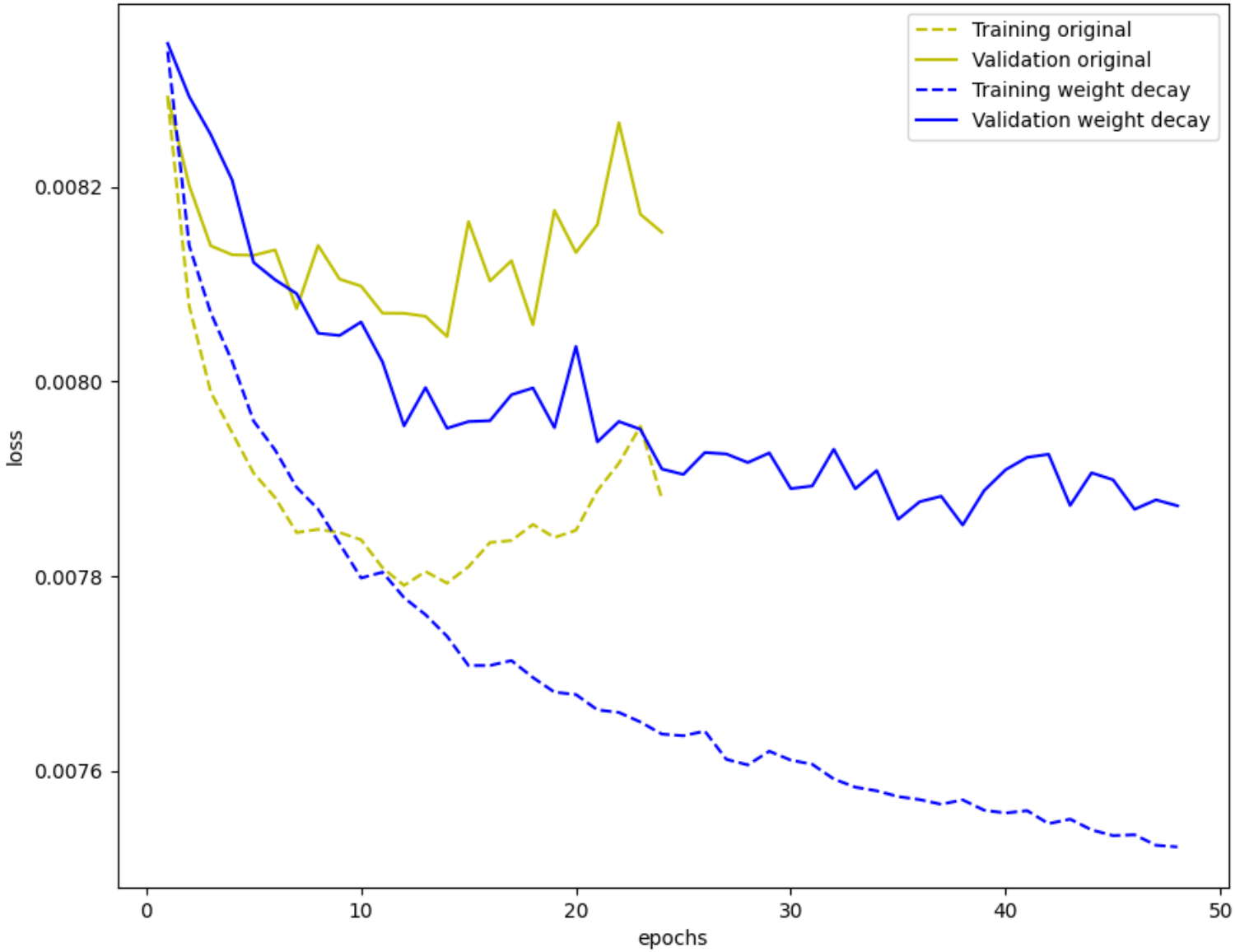
Example 4


0.0083
0.0082
0.0081
0.0080
0.0079
0.0078
loss
0
5
10
15
20
25
epochs
Train Loss
Validation Loss
Early Stopping Checkpoint

Example 4


0.0082
0.0080
0.0078
0.0076
loss
0
10
20
30
40
50
epochs
Train weight decay
Validation weight decay
Early Stopping Checkpoint

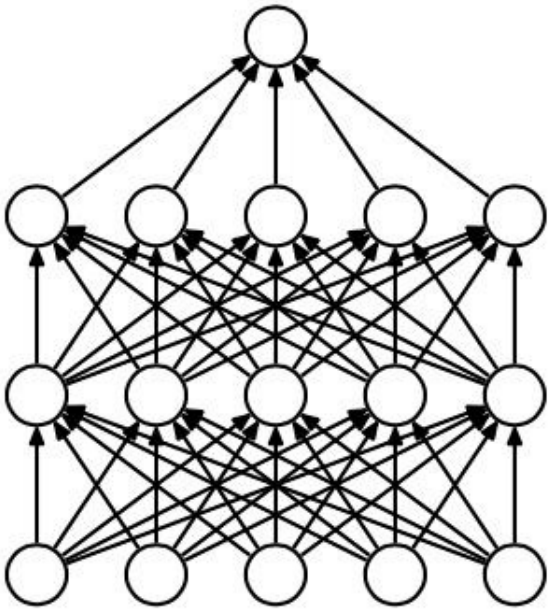
Example 4


Training original
Validation original
Training weight decay
Validation weight decay
0.0082
0.0080
0.0078
0.0076
loss
0
10
20
30
40
50
epochs

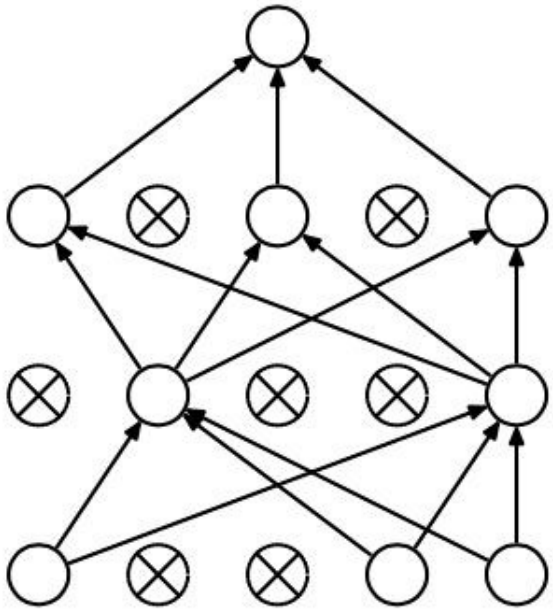
Dropouts

Overfitting can be avoided by training only a fraction of weights in each iteration. The key idea of ‘dropouts’ is to randomly drop neurons (along with their connections) from the networks during training.

This prevents neurons from co-adapting and thereby reduces overfitting.

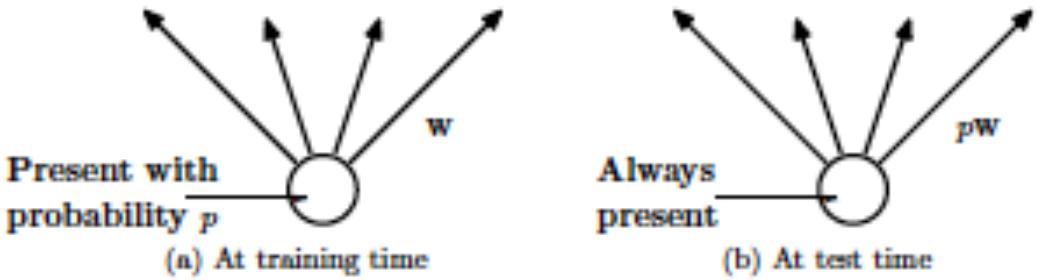


Fully connected network



Network with dropouts

Dropouts



At the training time, the units (neurons) are present with a probability p and presented to the next layer with weight $\boldsymbol{W}$ to the next layer.

This results in a scenario that at test time, the weights are always present, and presented to the network with weights multiplied by probability p. That is, The output at the test time is multiplied by $\frac{1}{p}$.

Applying dropouts result in a 'thinned network' that consists of only neurons that survived. This minimizes the redundancy in the network.

Dropout ratio

Dropout ratio is the fraction of neurons to be dropped out at one forward step.

Dropout ratio (p) has to be specified with nn.Dropout() in torch after activation function.

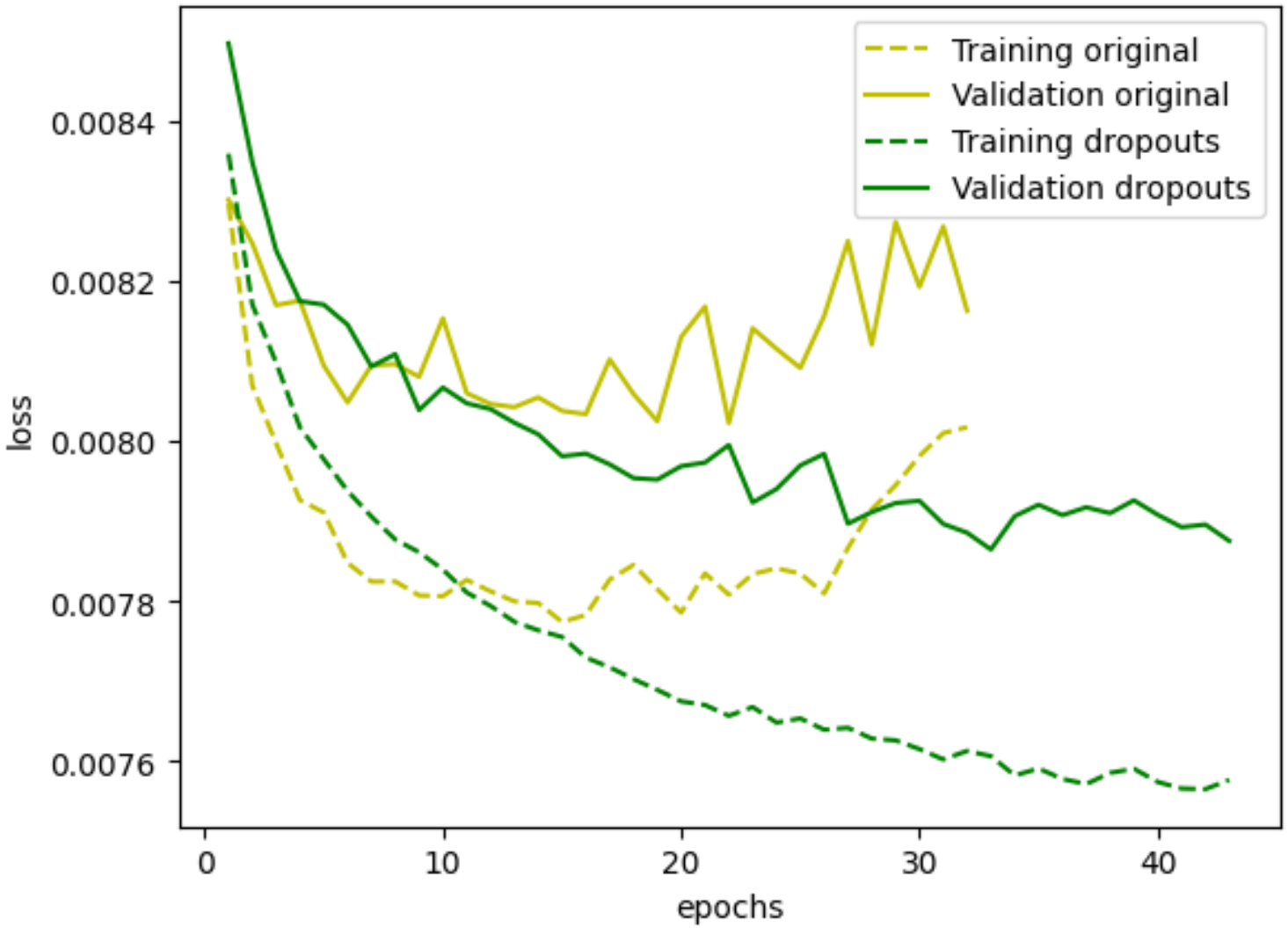
```
nn.Dropout(p=0.2)
```

Example 5: dropouts

When training with dropouts, we train on a thinned network dropping out units in each mini-batch.

```
class NeuralNetwork_dropout(nn.Module):
    def __init__(self, hidden_size = 100, drop_out=0.5):
        super(NeuralNetwork_dropout, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(32*32*3, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, hidden_size),
            nn.ReLU(),
            nn.Dropout(p=drop_out),
            nn.Linear(hidden_size, 10),
            nn.Softmax(dim=1)
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```

Example 5


Training original
Validation original
Training dropouts
Validation dropouts
0.0084
0.0082
0.0080
0.0078
0.0076
loss
0
10
20
30
40
epochs

Summary

- **Model selection**
  - Holdout method
  - Resampling methods
    - Random subsampling
    - K-fold cross-validation
    - LOO cross-validation
  - Three-way data split

- **Methods to overcome overfitting**
  - Early stopping
  - Weight regularization
  - Dropouts