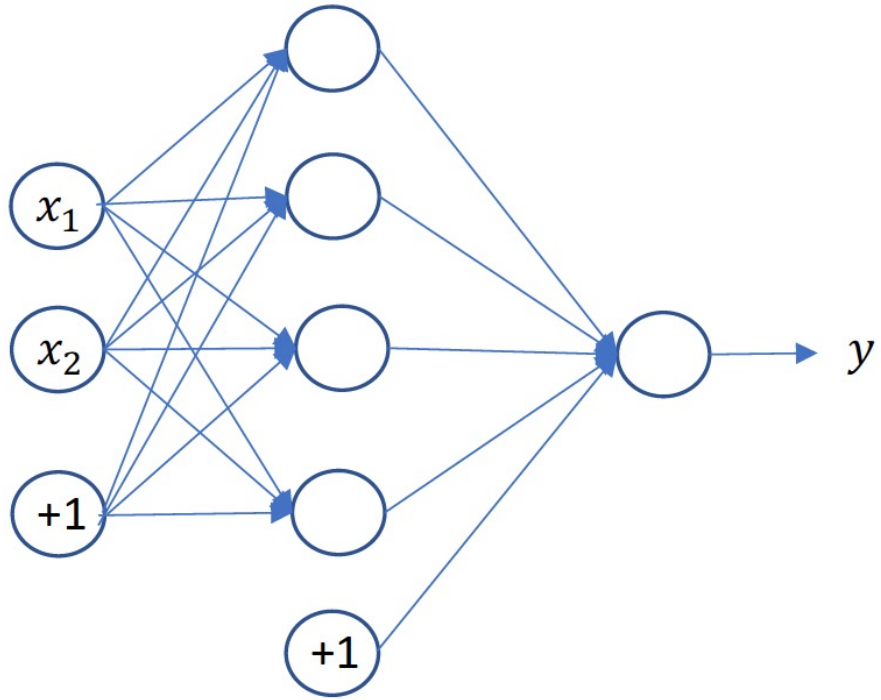
Model selection and overfitting

CS4001 – Tutorial 5



1. A feedforward network consisting of one hidden layer of perceptrons and a linear output neuron receives inputs two-dimensional inputs (x_1, x_2). Train the network to predict the following function:

$$y = sin(\pi x_1)cos(2\pi x_2)$$

where $-1.0 \leq x_1, x_2 \leq +1.0$.

Use data points distributed in an equally spaced 10x10 grid spanning the input space and the following procedures for training and testing:

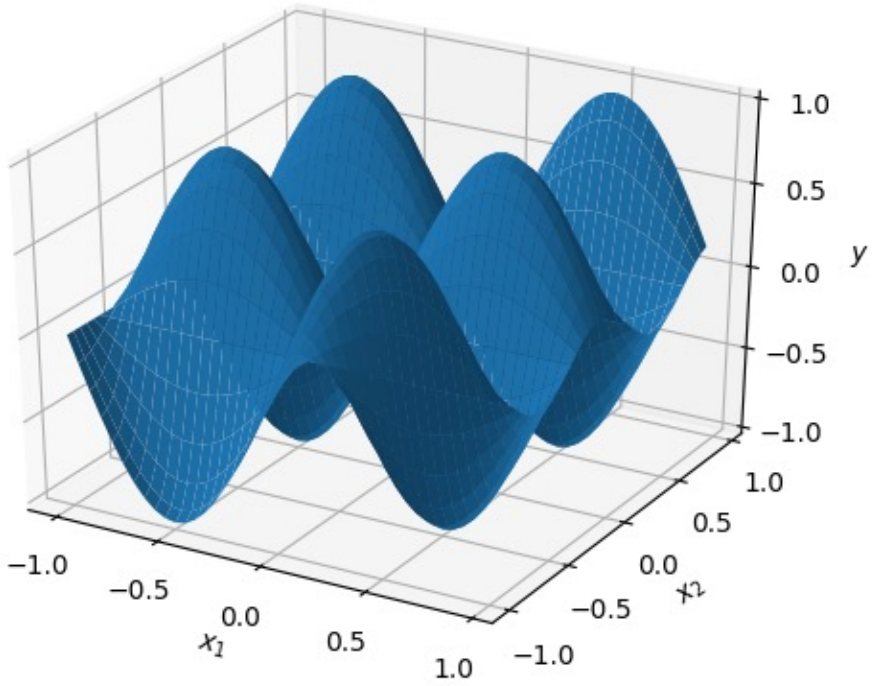
a) Random subsampling with 70:30 data-split for training and testing.
b) Five-fold cross validation
c) Three-way data split

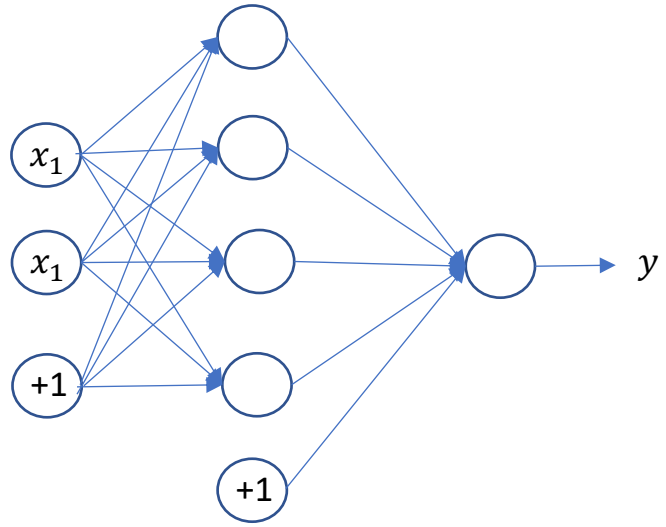
Repeat each procedure in 10 different experiments and determine the optimal number of hidden neurons in the space of $\{2, 4, 6, 8, 10\}$ and the error of the model.

Use a learning factor $\alpha = 0.05$ and early stopping to cease the learning epochs.

$y = sin(\pi x_1)\cos(2\pi x_2)$ where $-1.0 \leq x_1, x_2 \leq +1.0$.

Note that $-1.0 \leq y \leq +1$.

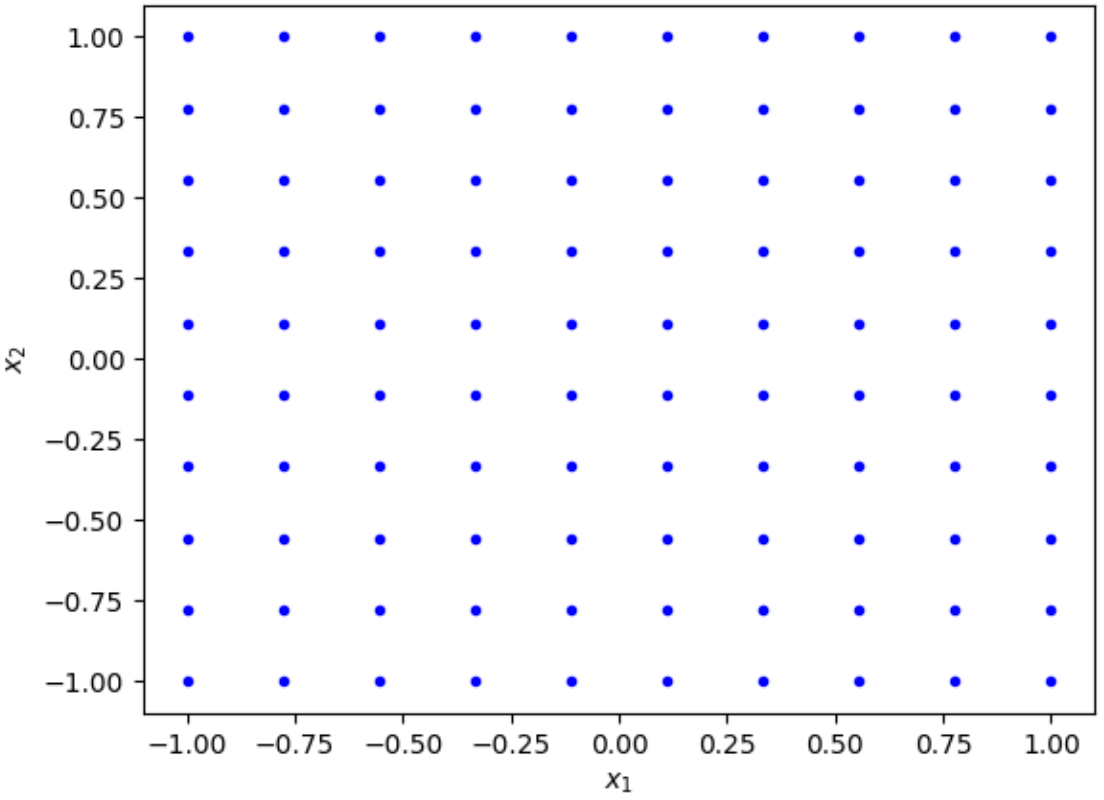


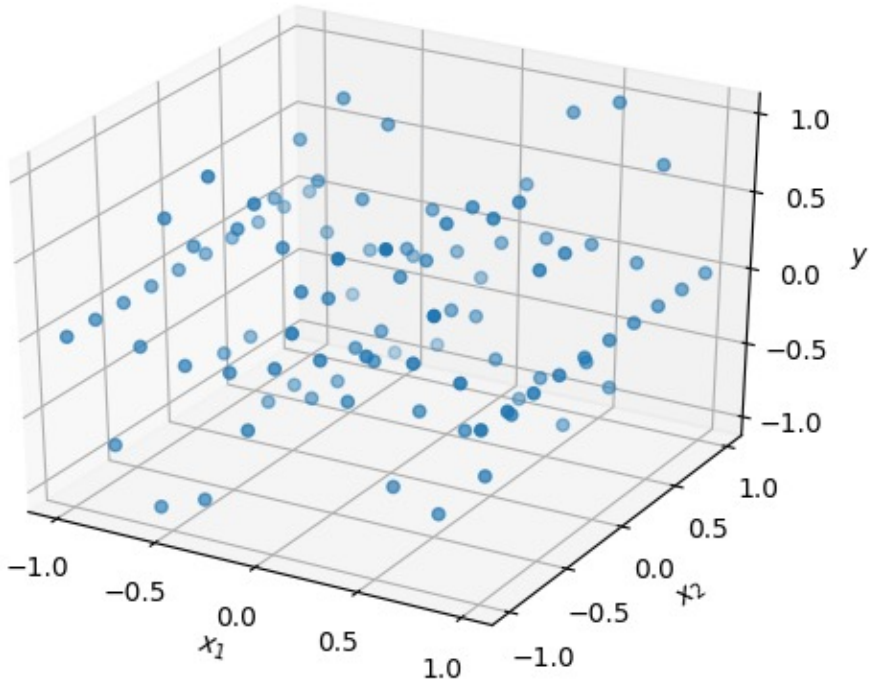


Hidden neurons perceptrons
the output neurons is a linear neuron.

$$y = sin(\pi x_1)\cos(2\pi x_2) \quad \text{where } -1.0 \leq x_1, x_2 \leq +1.0.$$

Training data is in an input grid of 10x10

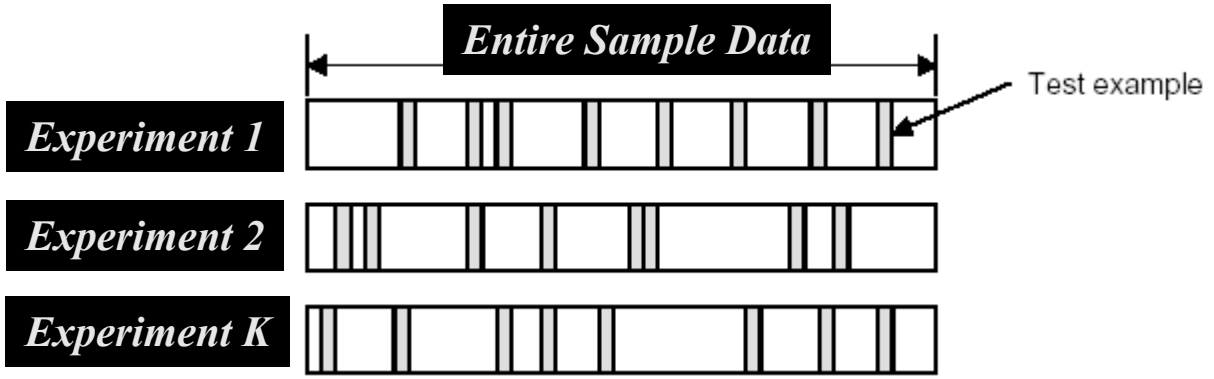




```
no_labels = 1
no_features = 2
no_exps = 10


class MLP(nn.Module):
    def __init__(self, no_features, no_hidden, no_labels):
        super().__init__()
        self.mlp_stack = nn.Sequential(
            nn.Linear(no_features, no_hidden),
            nn.Sigmoid(),
            nn.Linear(no_hidden, no_labels),
        )

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

```
class EarlyStopper:
    def __init__(self, patience=10, 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
```

K Data Splits: Random Subsampling



For each experiment k:

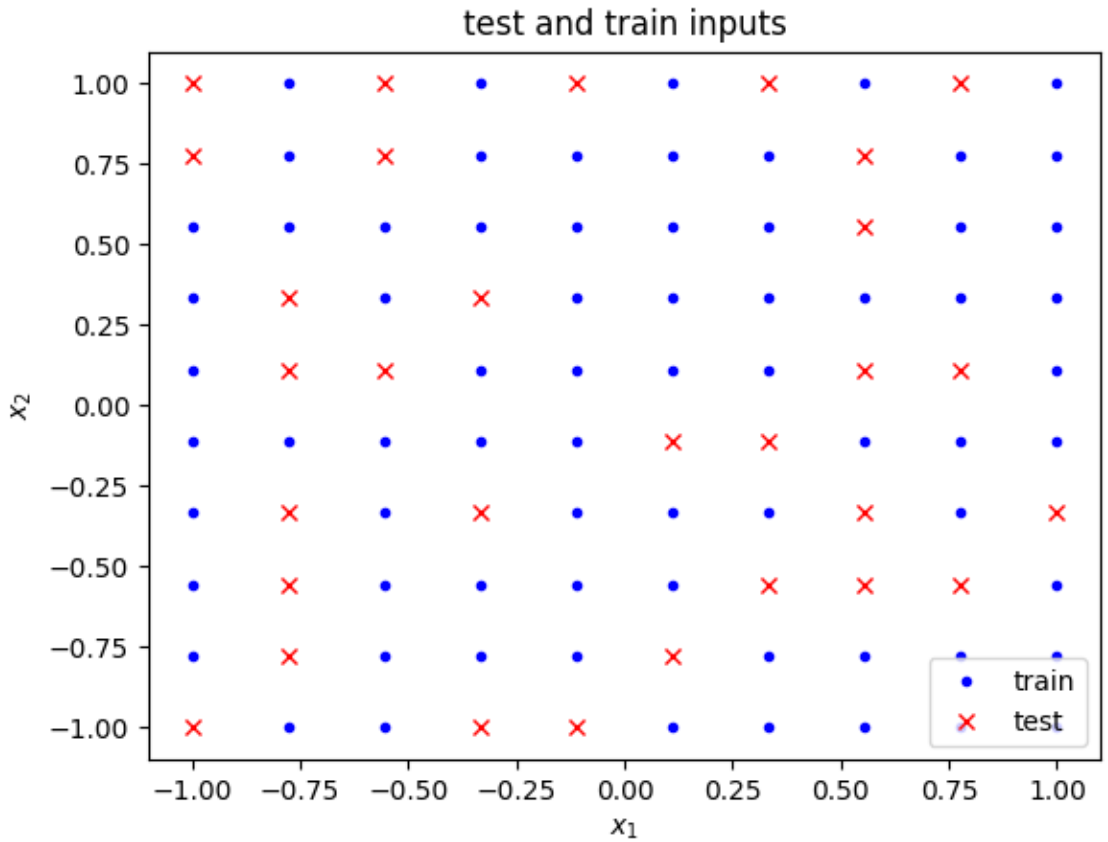
For each model m:

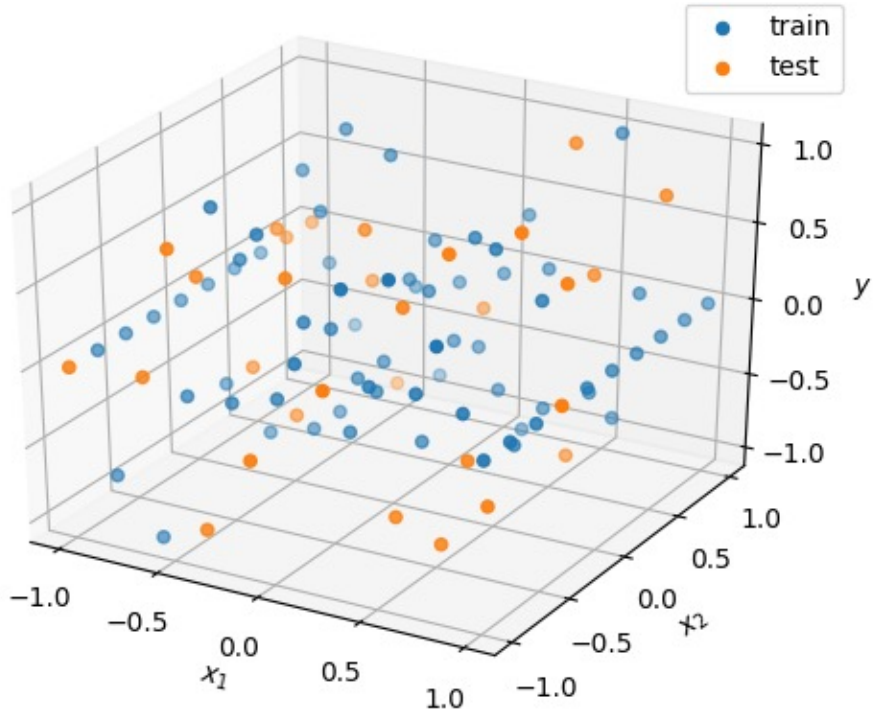
Compute error $e_{k,m}$

Compute mean error $e_m = \frac{1}{K}\sum_{k=1}^{K} e_{k,m}$

Optimal model $m^* = argmin\ e_m$

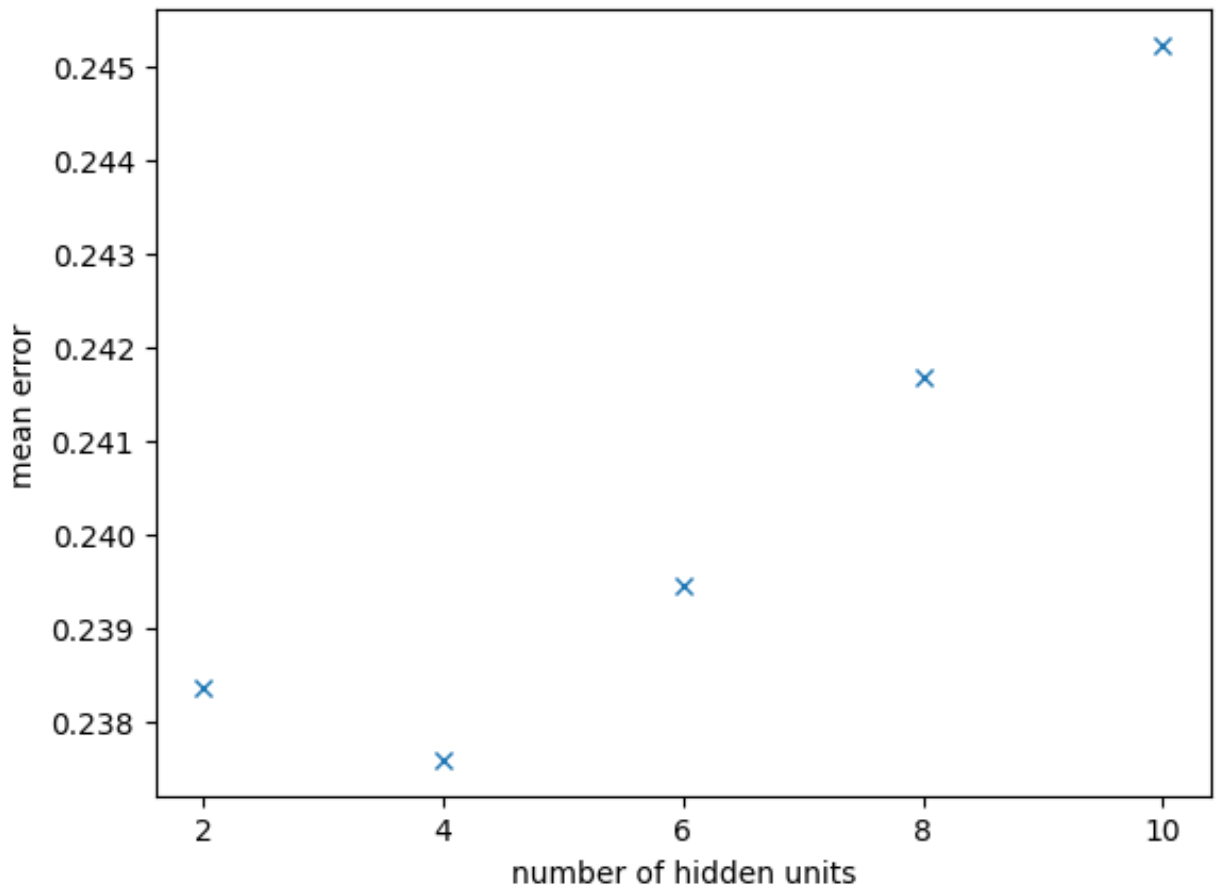
For each experiment, data is split into train and test data at [30: 70]





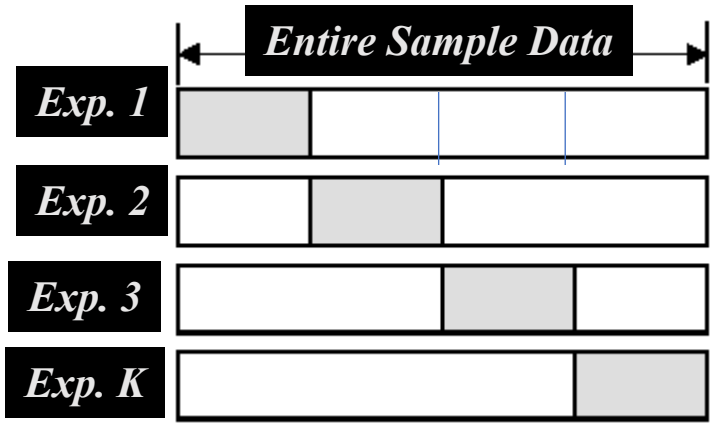
For each experiment, we build models with hidden neurons $\{2, 4, 6, 8, 10\}$

Mean error of 10 experiments



Optimum number of hidden neurons = **4**

K-fold Cross Validation



For every fold f:

For every model m

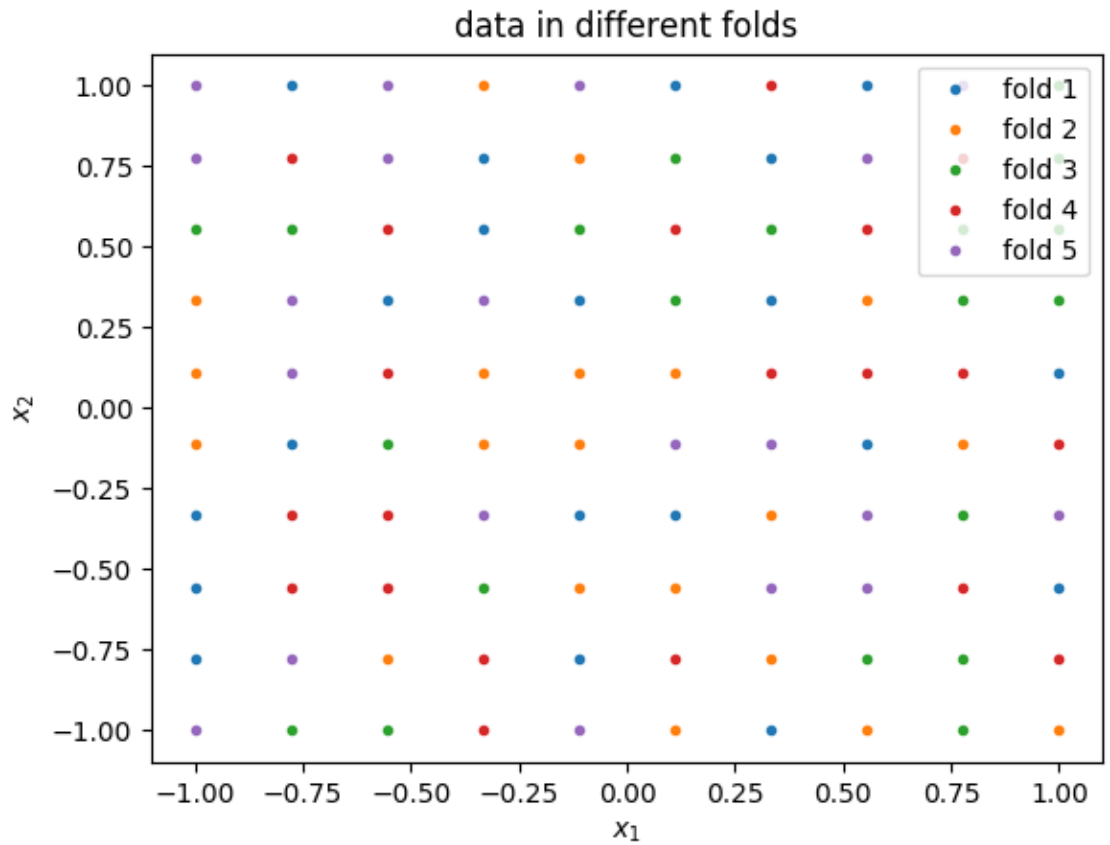
Train the model, using data not in fold f

Compute error $e_{m,f}$ on data in fold f

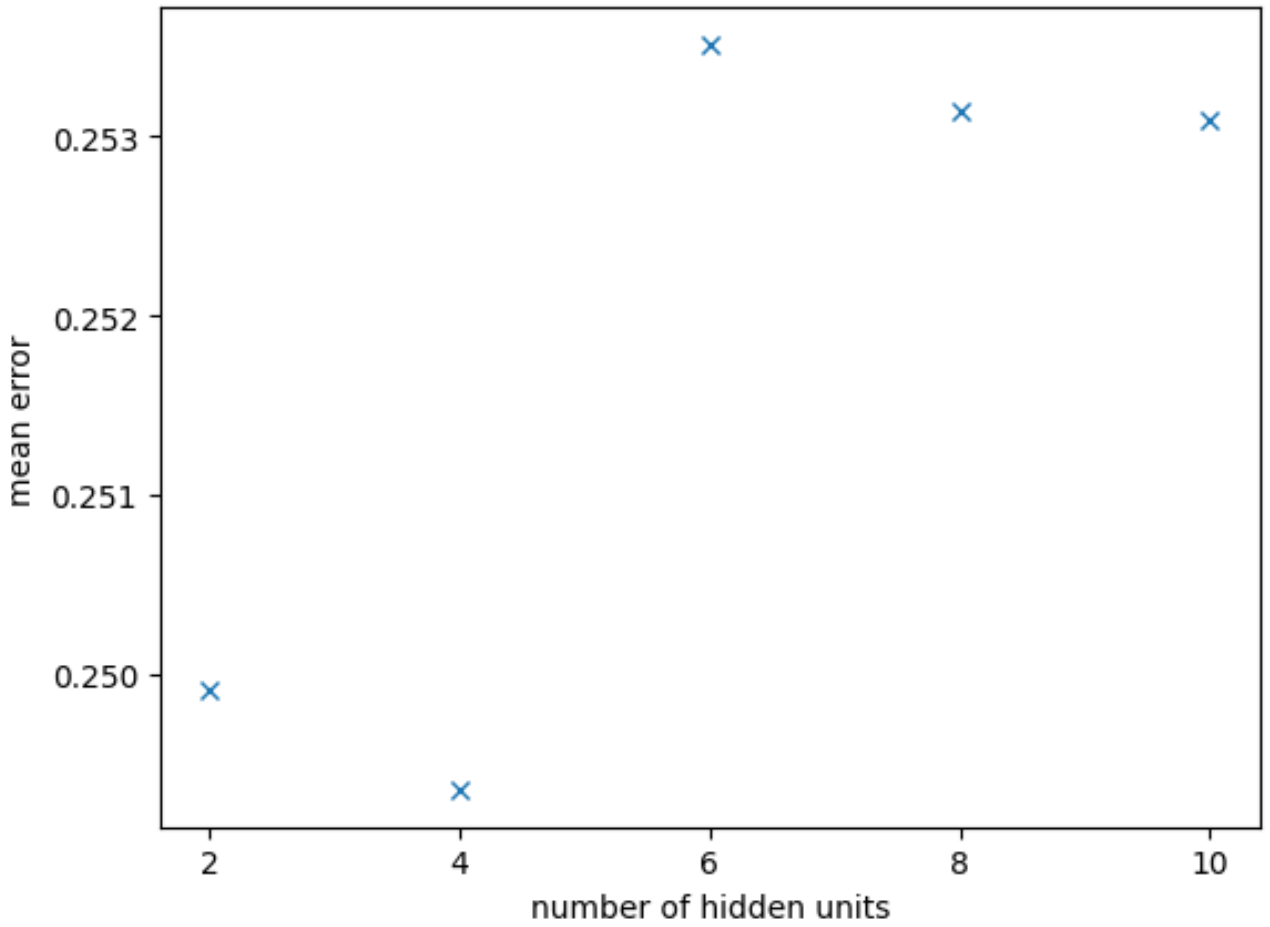
For every model m

CV error $e_m = \frac{1}{F}\sum_f e_{m,f}$

Select the model with minimum CV error, $m^* = argmin\ e_m$

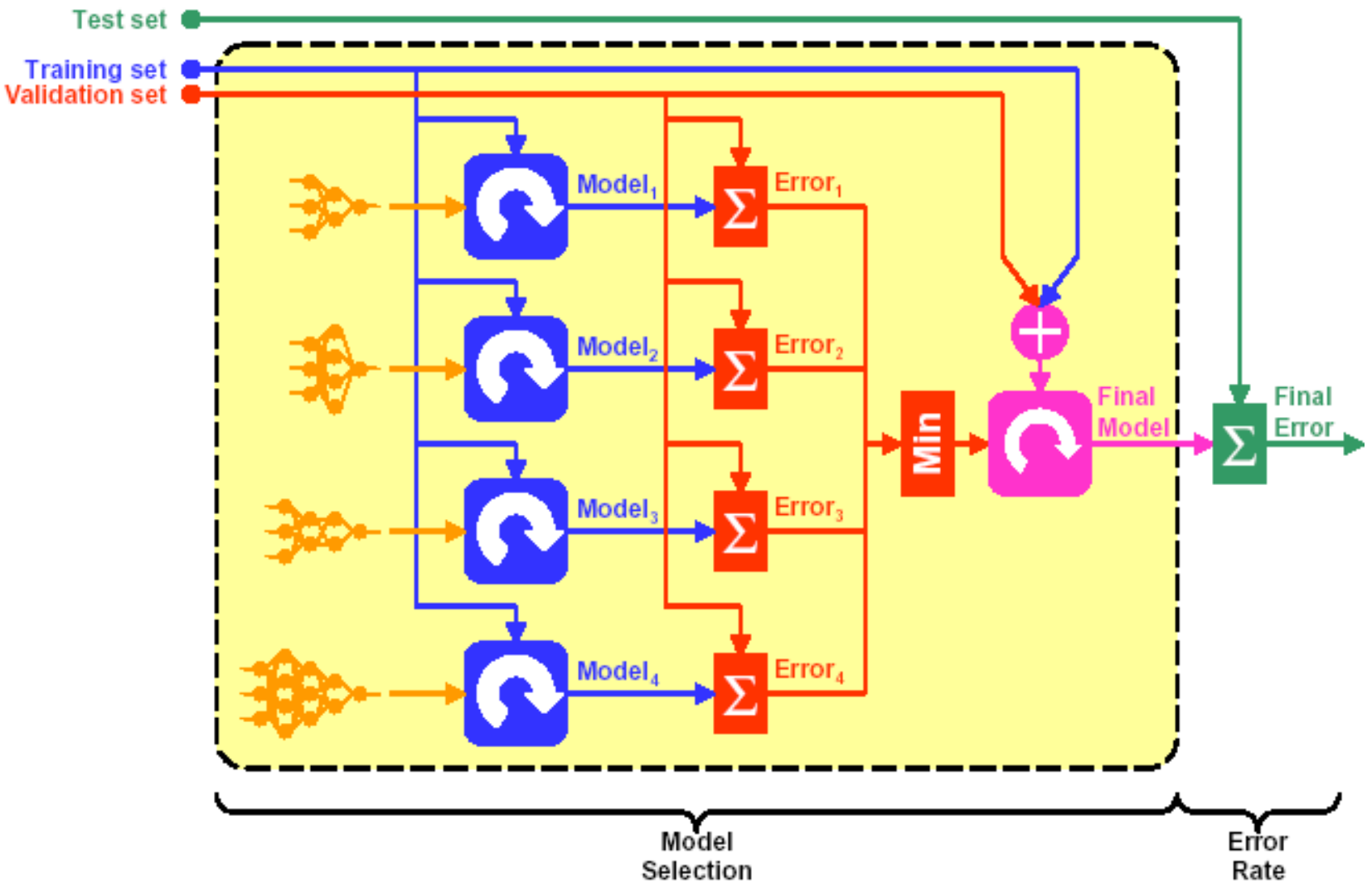

data in different folds
fold 1
fold 2
fold 3
fold 4
fold 5
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
x_2
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
x_1

Mean cross-validation error of 10 experiments



Optimum number of hidden neurons = **4**

Three-Way Data Splits Method



Three-Way Data Splits Method

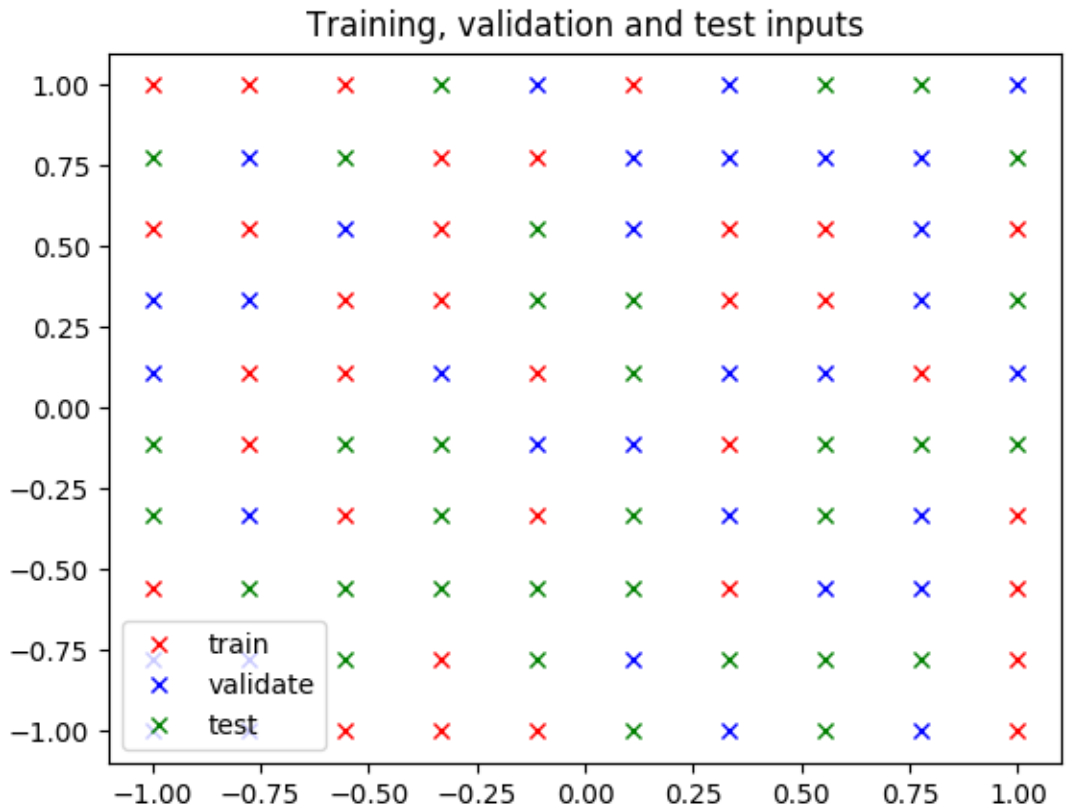
- **Training set**: Each model m, train the network and find the minimum error:

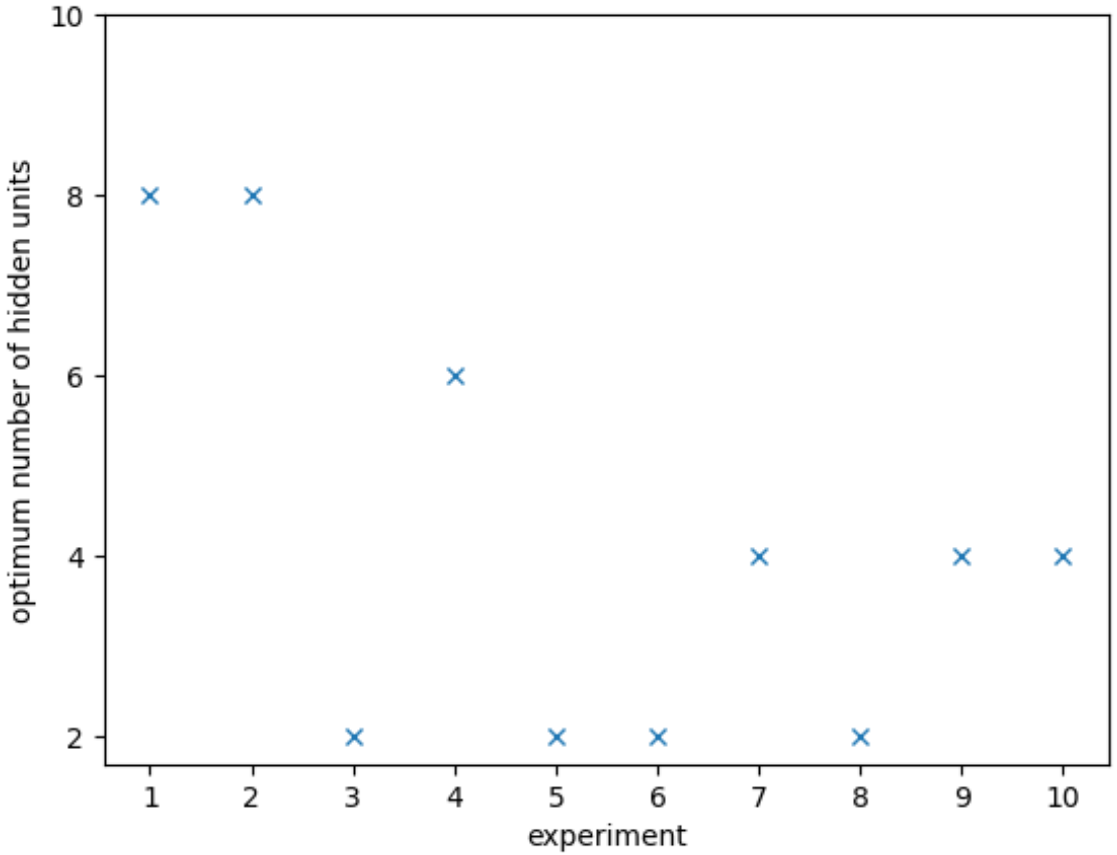
$$\boldsymbol{W}_m, \boldsymbol{b}_m = \underset{\boldsymbol{W},\boldsymbol{b}}{\operatorname{argmin}} J$$

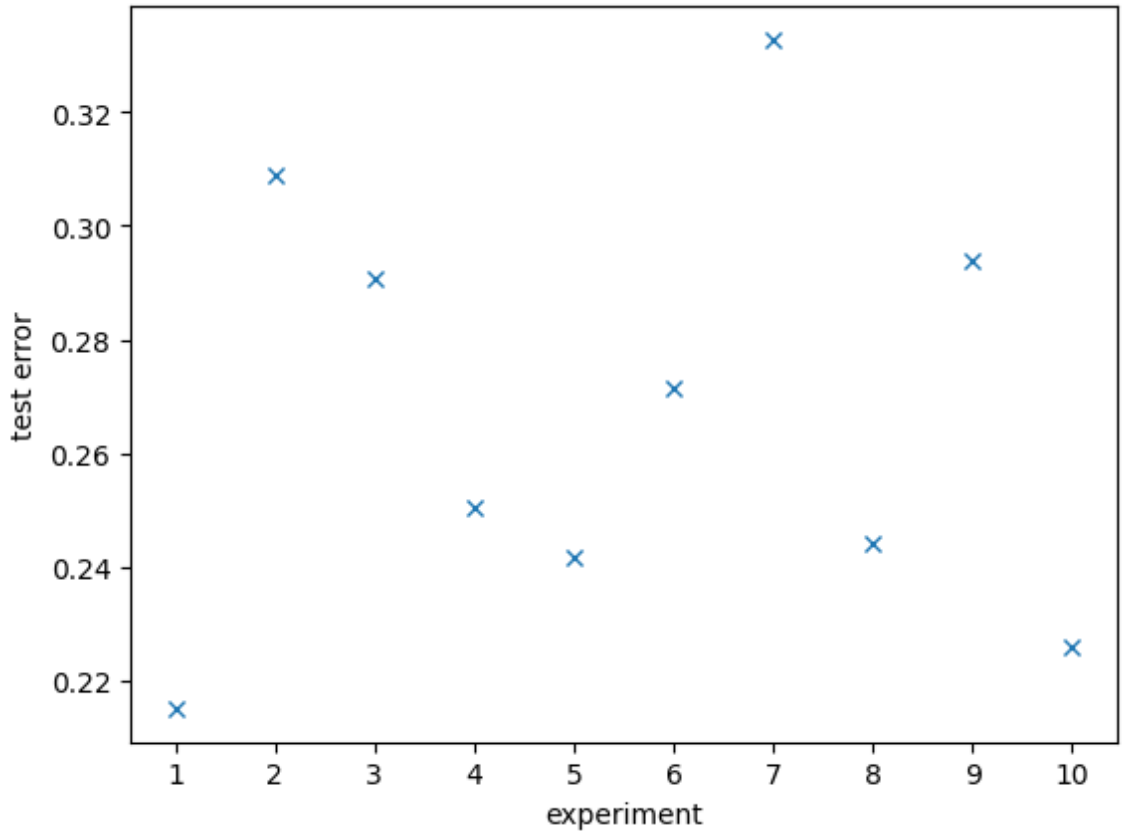
- **Validation set**: The optimal model m^* is given by

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

- **Training + Validation set:** Train the optimal model m^*
- **Test set**: determine test error on trained m^*


Training, validation and test inputs
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
train
validate
test





Optimal number of hidden neurons is 2 (Max number of times)
Average m.s.e.: 0.215 (by taking average of test error when hidden neurons = 2)

2. The CIFAR-10 dataset consists of 60,000 32x32 colour images from 10 classes, with 6,000 images per class:
https://www.cs.toronto.edu/~kriz/cifar.html

There are 50,000 training images and 10,000 test images. Read CIFAR-10 datasets from torchvision.datasets:

```
from torchvision.datasets import CIFAR10
```

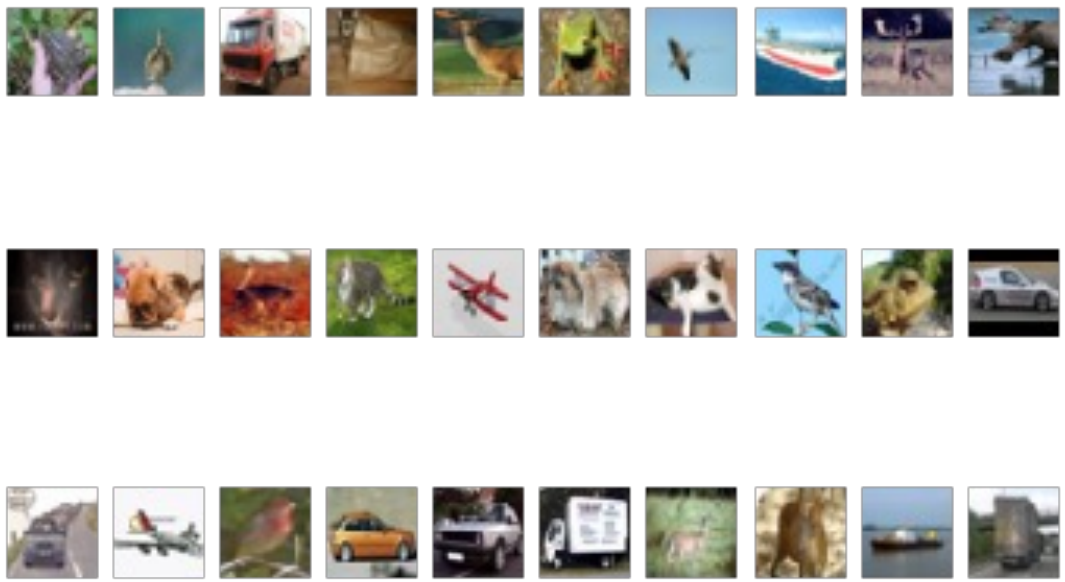
a) Build three DNN of different complexity with three hidden layers to classify the images:
   i. A **small network** with 50 neurons in every hidden layer
   ii. A **medium network** with 100 neurons in every hidden layer
   iii. A **large network** with 500 neurons in every layer

Plot cross-entropies against epochs for train and test datasets. Use the Adams optimizer with default parameter for training and early stopping criterion to terminate. Comment on the overfitting and underfitting of the networks if any.

The CIFAR-10 dataset: https://www.cs.toronto.edu/~kriz/cifar.html

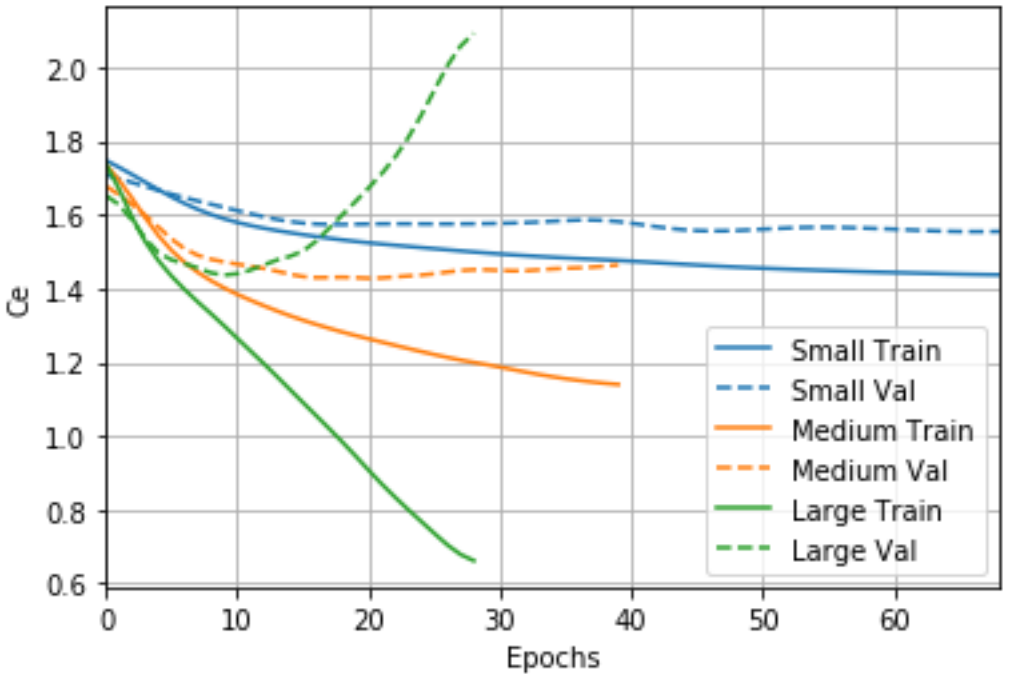
50000 training and 10000 test images; Images are 32x32 colour images

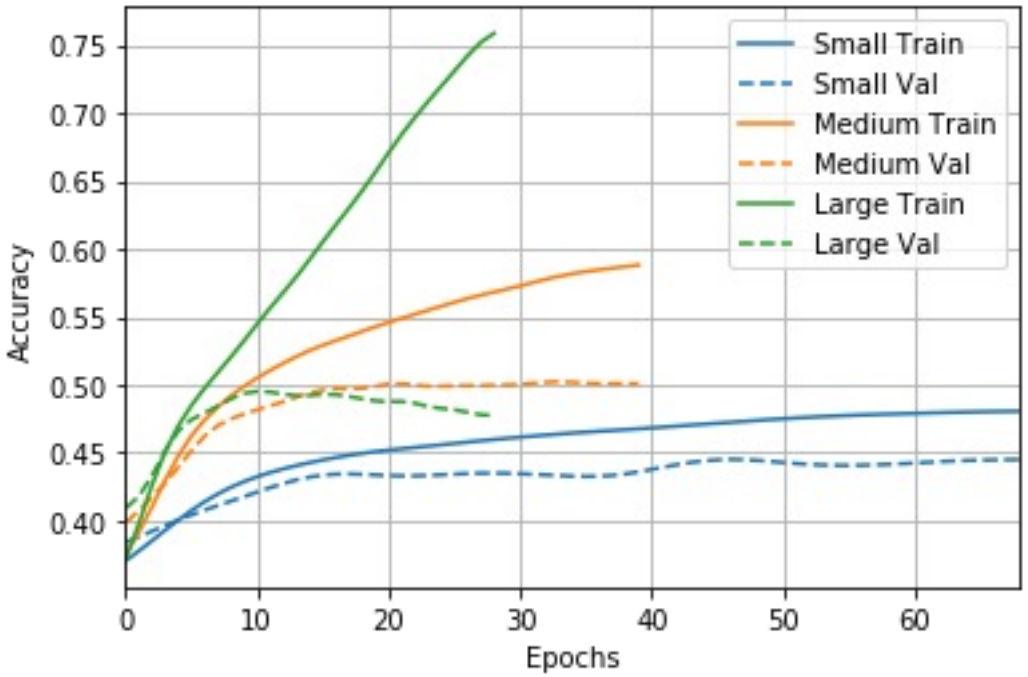
from torchvision.datasets **import** CIFAR10



```
class NeuralNetwork(nn.Module):
    def __init__(self, hidden_size=100):
        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, 10),
            nn.Softmax(dim=1)
        )

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

```
model = NeuralNetwork(hidden_size=50)

loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters())
early_stopper = EarlyStopper(patience=10, min_delta=0)



for t in range(max_epochs):
    train_loss, train_correct = train_loop(train_dataloader, model, loss_fn, optimizer)
    test_loss, test_correct = test_loop(test_dataloader, model, loss_fn)

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





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

b) Using the **large network**, demonstrate how the following methods applied to all the layers could overcome overfitting of the network:

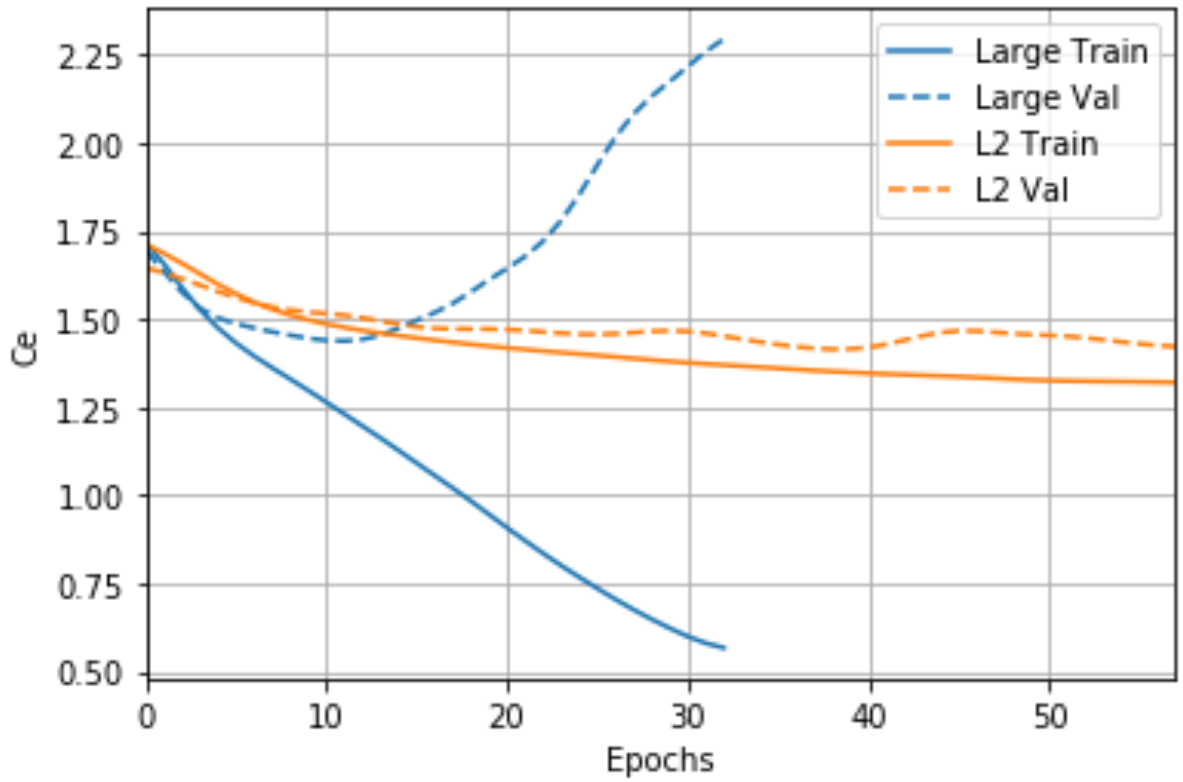
i. Weight regularization with L2 norm. Use weight decay parameter $\beta = 0.001$.

ii. Dropouts at a probability $p = 0.5$.

iii. Combining both weight regularization and dropouts from (i) and (ii), respectively.

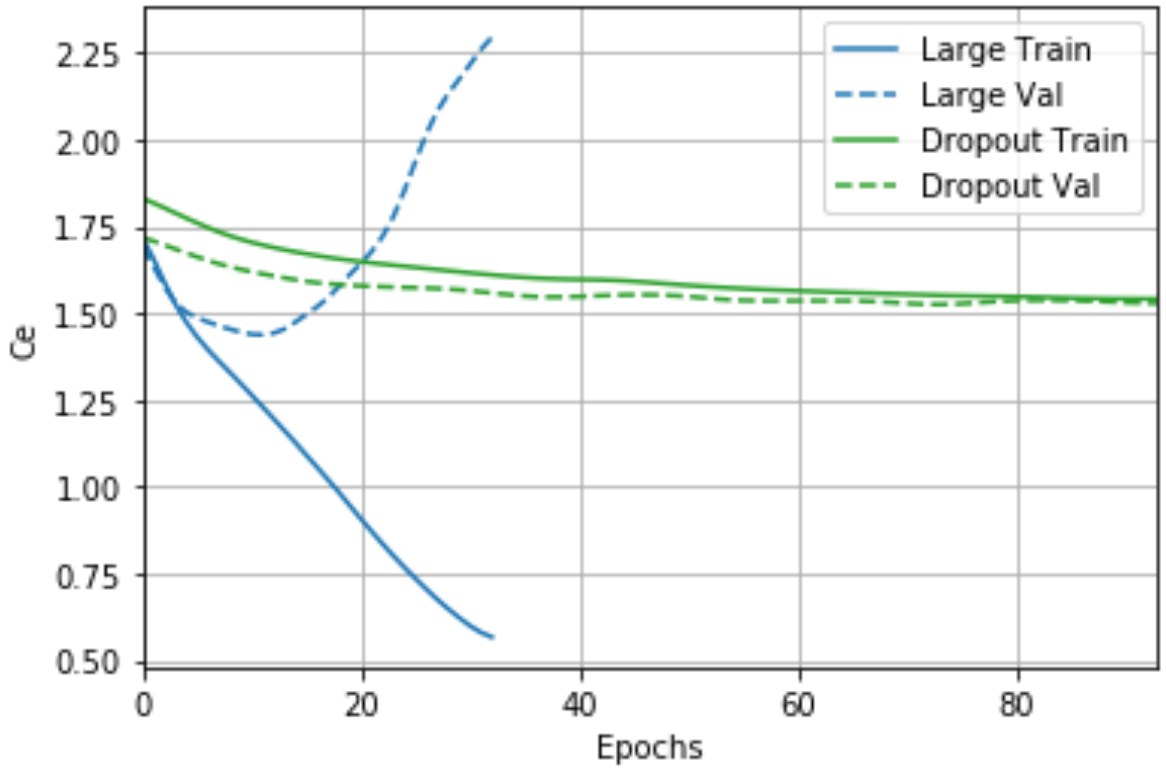
L2 weight regularization

optimizer = torch.optim.Adam(model.parameters(), weight_decay=0.001)



Dropouts

```
class NeuralNetwork_dropout(nn.Module):
    def __init__(self, hidden_size = 500, 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.Dropout(p=drop_out),
            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
```



Combined: L2 regularization + dropouts