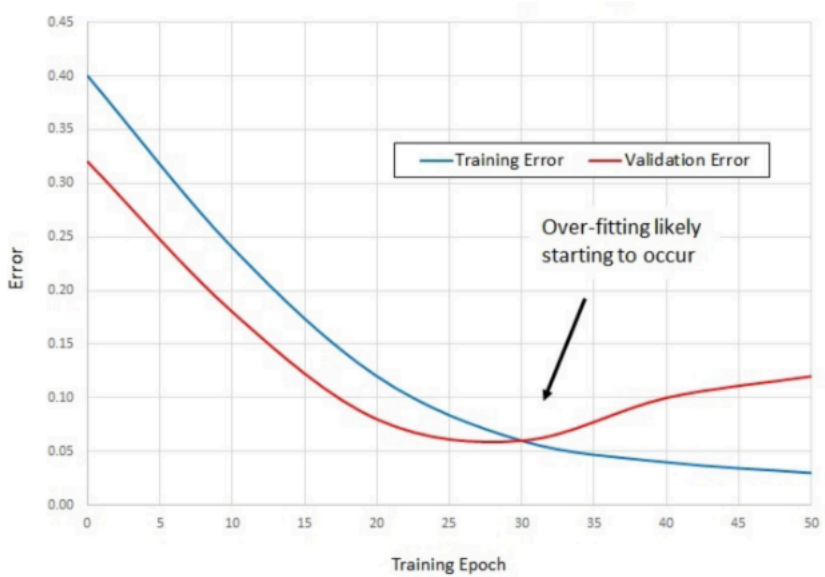
Optimization Vs. Generalization

- To find the best parameters for our model, we mimimize the average loss over all training examples.
    - This is an optimization problem.
- Minimizing the training loss is **not** however our ultimate objective:
    - What we want is our model is to **generalize** i.e. to exhibit good accuracy on a seperate validation/test set.
- The following figure show this (note we plot the error = 1 - accuracy):



Checking with a Validation Set

- The final solution depends on:
  - the **initial values of the parameters**.
  - the **optimization algorithm** chosen (e.g. SGD, Adam).
  - the **hyper-parameters** chosen (e.g. batch-size, learning rate).
- Also we want to avoid overfitting:
  - We can use **regularization**.
- All of the above need to be chosen accordingly every time we train a neural network for a new task.
  - The **architecture** is also important (will not be covered here).
- Some choices can be checked by monitoring the training loss.
- However, we always need to check using a **seperate validation set**.

Parameter Initialization

Default Initialization

- Previously we used a normal distribution to initialize the values of our weights.
- If we do not specify the initialization method, PyTorch will use a default random initialization method, which often works well in practice
    - He initialization
- `torch.nn.init` package provides several ways to init the parameters.

Xavier Initialization

- As illustrated by Glorot and Bengio, a good initialization of the weights could be the one such that the values of the input features and the output features are in the same range.
- To achieve this, Xavier initialization initializes weights from a Gaussian with zero mean and variance $\sigma^2 = \frac{2}{n_{\text{in}}+n_{\text{out}}}$.
- PyTorch implementation: `torch.nn.init.xavier_normal_`
- Xavier's initialization can be applied when sampling the weights from a uniform distribution.
    - The uniform distribution $U(-a, a)$ has variance $\frac{a^2}{3}$.

$$U\left(-\sqrt{\frac{6}{n_{\text{in}}+n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}}+n_{\text{out}}}}\right).$$

He Initialization

- Xavier initialization assumes non-existence of nonlinearities.
    - This is easily violated in neural networks.
- He initialization takes that into account.
- He initialization initializes weights from a Gaussian with zero mean and variance $\sigma^2 = \frac{2}{n_{\text{in}}}$.
- PyTorch implementation: `torch.nn.init.kaiming_normal_`

Regularization

- Regularisation is a standard technique to reduce overfitting in Machine Learning. Regularisation techniques for NNs include:
    - Early Stopping
    - L2 regularization (weight decay)
    - Dropout
    - Data augmentation
    - More exotic ones (e.g. mix-up)

Early Stopping

- Simply stop training when accuracy starts dropping on a seperate validation set

Weight decay (L2 regularization)

- **Weight Decay** (commonly called L_2 regularization), might be the most widely-used technique for regularizing Machine Learning models.
- WD adds $\|\mathbf{w}\|^2$ as an additional term to loss function

$$L(\mathbf{w}, b) + \frac{\lambda}{2}\|\mathbf{w}\|^2,$$

  - For $\lambda = 0$, we recover the original loss function $L(\mathbf{w}, b)$.
  - For $\lambda > 0$, we put a penalty on the magnitude of $\|\mathbf{w}\|$.
- The L_2 norm places a penalty on large components of the weight vector.
  - This biases our learning algorithm towards models that distribute the weights evenly across features.
  - In practice, this makes them more robust to measurement error in a single variable.
- Other norms are possible, e.g. the L_1 norm.
  - L_1 penalty lead to models that most weights are 0.
  - This can be used for *feature selection*, which may be desirable for some applications.

Weight decay Implementation

In [31]:
```
# Scratch implementation
# We need to iterate through all params
def l2_penalty(w):
    return torch.sum(w.pow(2)) / 2

model = torch.nn.Linear(10,10, bias=False)
reg_loss = 0
for param in model.parameters():
    reg_loss += l2_penalty(param)
```

In [32]:
```
# Pytorch's implementation
wd, lr = 0.0005, 0.1
model = torch.nn.Linear(10,10)
optimizer = torch.optim.SGD(model.parameters(), weight_decay=wd, lr=lr)
```

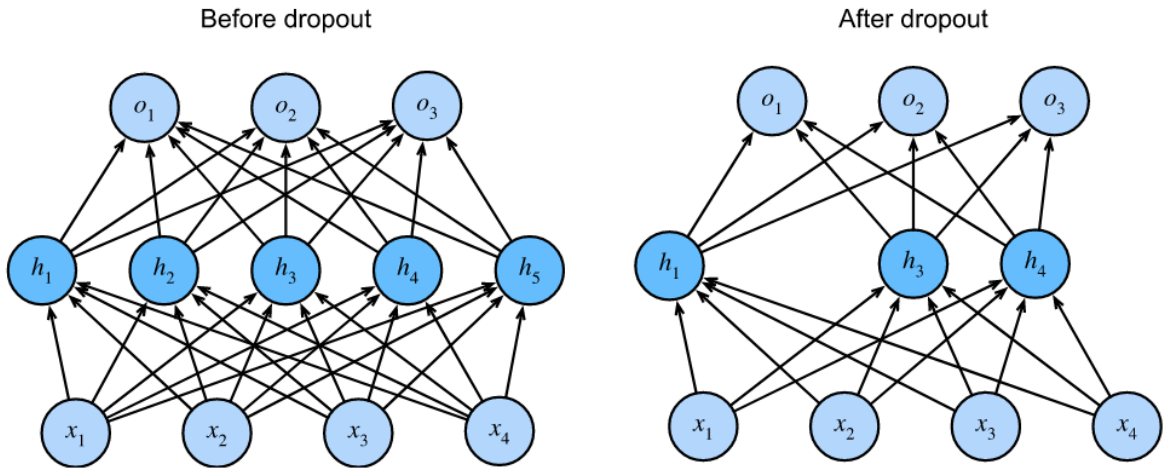
Droupout

- Neural network overfitting is characterized by a state in which each layer relies on a specifc pattern of activations in the previous layer,
    - This is called *co-adaptation*.
- Dropout is a method to break this co-adaptation.
- During training neurons are randomly dropped out: with *dropout probability* p, each intermediate activation h is replaced by a random variable h' as follows:

$$h' = \begin{cases} 0 & \text{with probability } p \\ \frac{h}{1-p} & \text{otherwise} \end{cases}$$

Droupout

- In the example below h_2 and h_5 are dropped.
- The calculation of the outputs no longer depends on h_2 or h_5 and their respective gradient also vanishes when performing backpropagation.
- In this way, the calculation of the output layer cannot be overly dependent on any one element of $h_1, \ldots, h_5$.



- Typically, we disable dropout at test time.
    - Given a trained model and a new example, we do not drop out any nodes.
- Some researchers use dropout at test time as a heuristic for estimating the *uncertainty* of neural network predictions.
    - If the predictions agree across many different dropout masks, then we might say that the network is more confident.

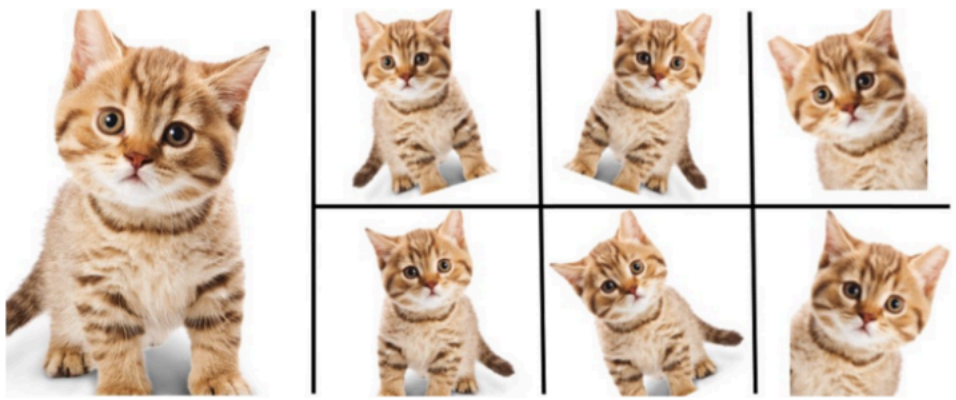
Dropout Implementation

In [33]:
```python
# implementation from scratch
def dropout_layer(x, dropout): #x is feature tensor
    assert 0 <= dropout <= 1
    # In this case, all elements are dropped out
    if dropout == 1:
        return torch.zeros_like(x)
    # In this case, all elements are kept
    if dropout == 0:
        return x
    mask = (torch.Tensor(x.shape).uniform_(0, 1) > dropout).float()
    return mask * x / (1.0 - dropout)
```

- PyTorch provides `torch.nn.Dropout(p)`
    - At test time remember to call `model.eval()`
- Dropout layer can be used after the ReLU non-linearity in each layer

Data Augmentation

- In general we will train our network for a large number of epochs.
    - Each epoch we will see a training example once.
- From epoch to epoch what we can do is not to use exactly the same example but apply some sort of noise to it so that it looks a bit different.
- This is to some extent equivalent to using more data during training
- For images we can apply several types of noise including scaling, rotation, colour jittering
- An example of geometric augmentation:



Learning Rate

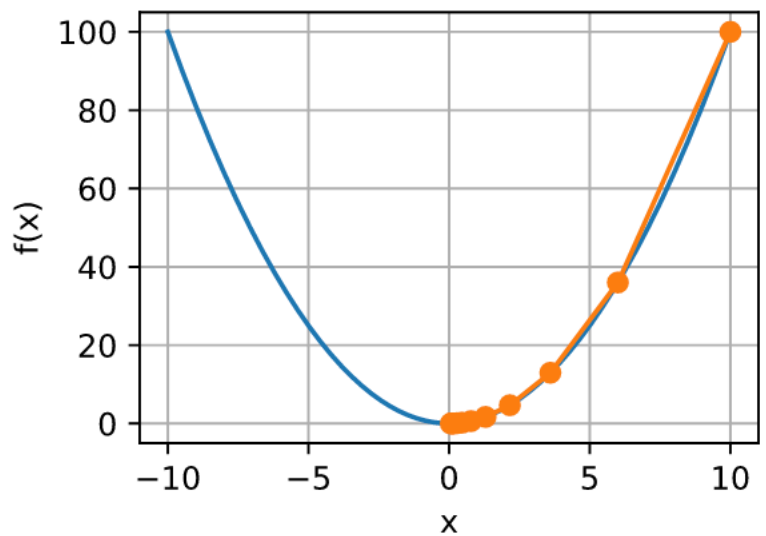
- Adjusting the learning rate is imporant to find good solutions.
- For simplicity we will consider gradient descent in one dimension.
- Example: consider the objective function $f(x) = x^2$.
- We use $x = 10$ as the initial value and assume $\eta = 0.2$.
- Using GD to update x for 10 times we see that, eventually, the value of x approaches the optimal solution.

In [24]:
```
f = lambda x: x**2  # Objective function
gradf = lambda x: 2 * x  # Its derivative

def gd(eta):
    x = 10.0
    results = [x]
    for i in range(10):
        x -= eta * gradf(x)
        results.append(float(x))
    print('epoch 10, x:', x)
    return results

res = gd(0.2)
```

```
epoch 10, x: 0.060466175999999997
```

In [25]:

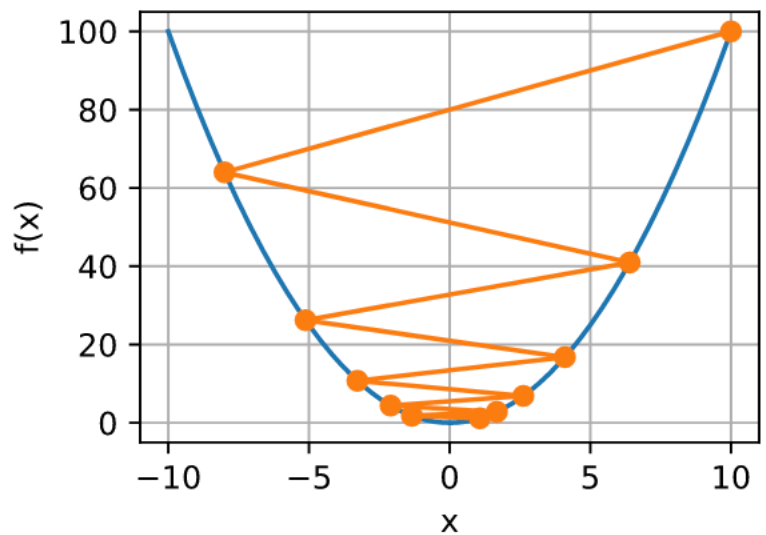
```
def show_trace(res):
    n = max(abs(min(res)), abs(max(res)))
    f_line = torch.arange(-n, n, 0.01)
    mu.set_figsize()
    mu.plot([f_line, res], [[f(x) for x in f_line], [f(x) for x in res]],
            'x', 'f(x)', fmts=['-', '-o'])

show_trace(res)
```



- We should try to use a high learning rate.
- This leads to both good solutions and faster convergence.
- If the learning rate is too high we might have unecessary oscillations close to the minumum.
    - Hence, we may want to reduce it after a number of epochs.

In [28]:
```
res = gd(0.9)
show_trace(res)
```

```
epoch 10, x: 1.0737418240000007
```



- Too high a learning rate results in divergence.

In [29]:
```
res = gd(1.0)
show_trace(res)
```

```
epoch 10, x: 10.0
```