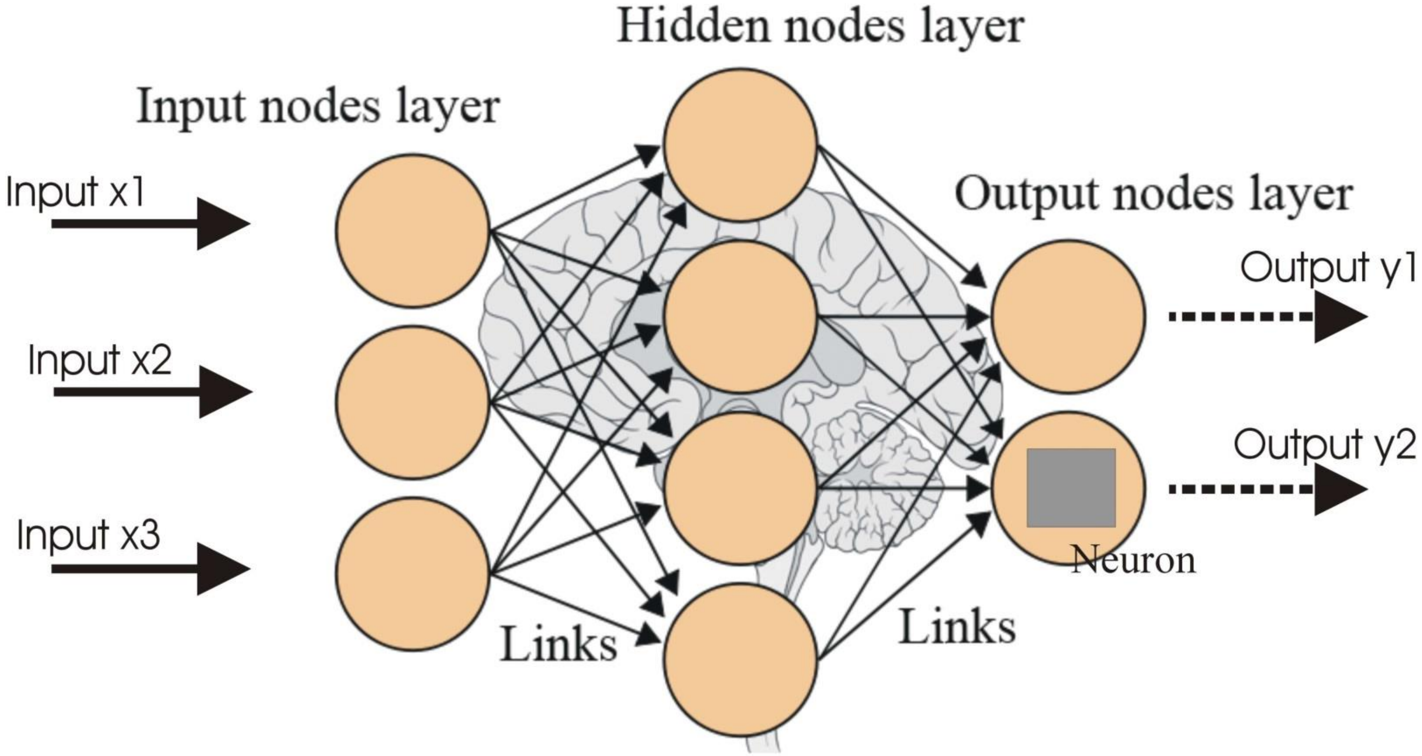
Inferencing Issues:
Generalization and Overfitting

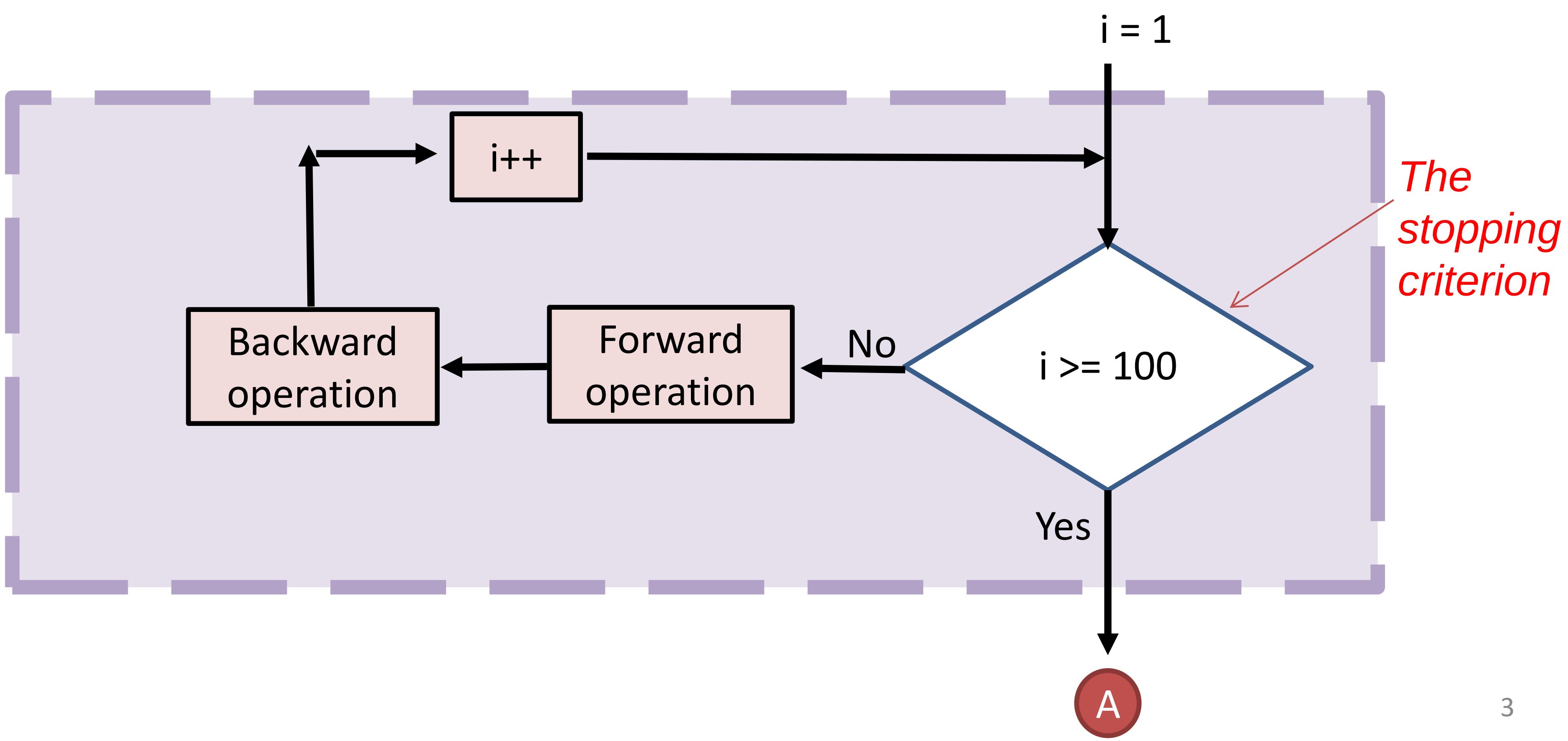
國立政治大學 資訊管理學系
蔡瑞煌 特聘教授

2-Layer nets; SLFN



Where we are now...

The algorithm without extra stopping criteria



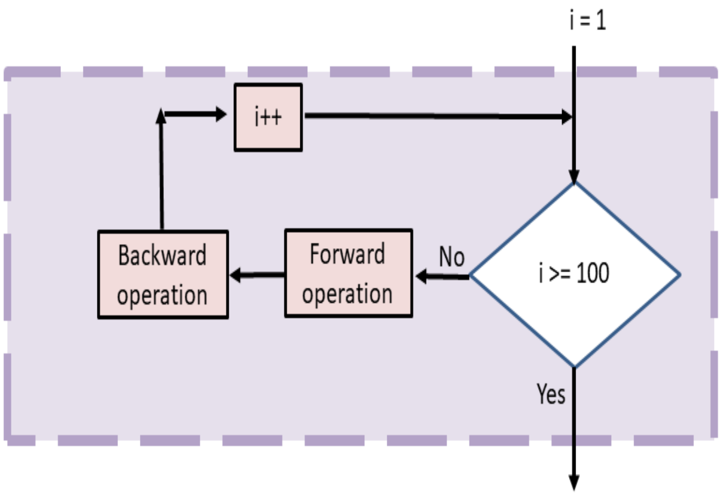
Learning codes of SLFN

- PyTorch: cs231n 2020 Lecture 6-57
- PyTorch: cs231n 2020 Lecture 6-65
- TensorFlow 2.0+ vs. pre-2.0: cs231n 2020 Lecture 6-91
- TensorFlow: cs231n 2020 Lecture 6-101
- TensorFlow with optimizer: cs231n 2020 Lecture 6-102
- TensorFlow with optimizer & predefined loss: cs231n 2020 Lecture 6-103
- Keras: cs231n 2020 Lecture 6-104
- Keras: cs231n 2020 Lecture 6-106 (help handle the training loop)

Where we are now...

PyTorch: nn

Higher-level wrapper for working with neural nets

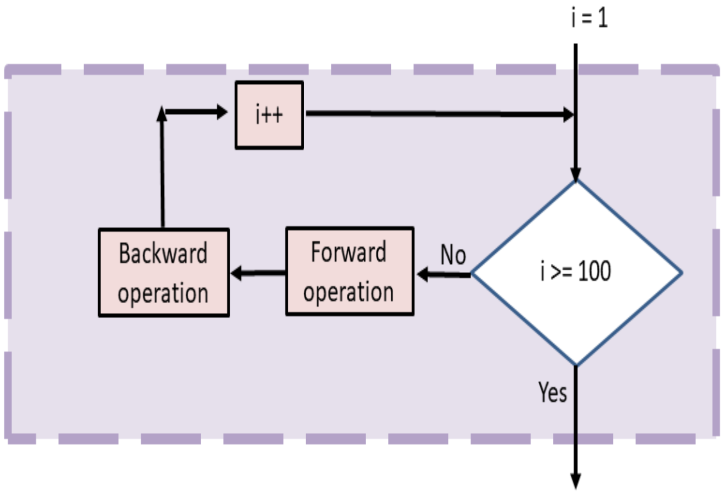



```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

model = torch.nn.Sequential(
          torch.nn.Linear(D_in, H),
          torch.nn.ReLU(),
          torch.nn.Linear(H, D_out))

learning_rate = 1e-2
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)

    loss.backward()

    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * param.grad
    model.zero_grad()
```

Where we are now...

PyTorch: nn
Define new Modules

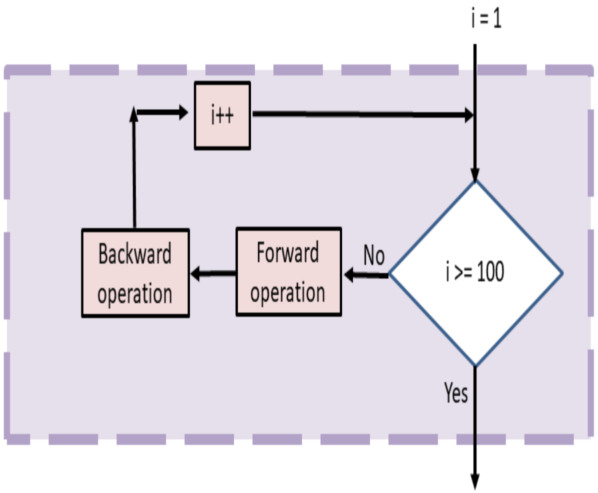
A PyTorch **Module** is a neural net layer; it inputs and outputs Tensors




```
import torch

class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

model = TwoLayerNet(D_in, H, D_out)

optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)

    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

Where we are now...

TensorFlow: Neural Net

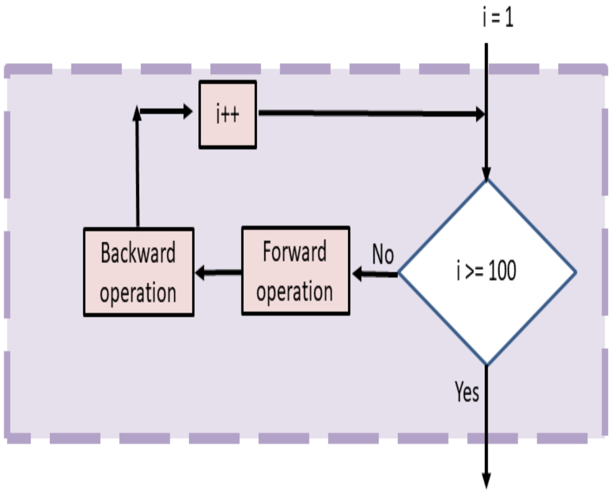



```python
N, D, H = 64, 1000, 100

x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
w1 = tf.Variable(tf.random.uniform((D, H)))  # weights
w2 = tf.Variable(tf.random.uniform((H, D)))  # weights

learning_rate = 1e-6
for t in range(50):
  with tf.GradientTape() as tape:
    h = tf.maximum(tf.matmul(x, w1), 0)
    y_pred = tf.matmul(h, w2)
    diff = y_pred - y
    loss = tf.reduce_mean(tf.reduce_sum(diff ** 2, axis=1))
  gradients = tape.gradient(loss, [w1, w2])
  w1.assign(w1 - learning_rate * gradients[0])
  w2.assign(w2 - learning_rate * gradients[1])
```

Where we are now...

TensorFlow: Loss

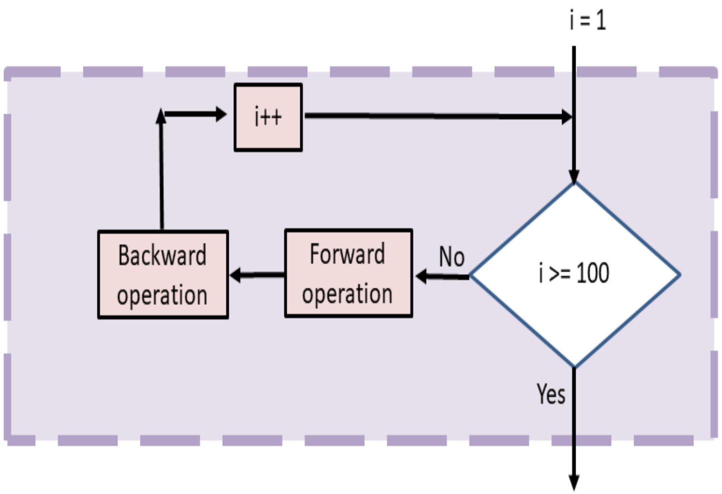
Use predefined common losses




```
N, D, H = 64, 1000, 100

x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
w1 = tf.Variable(tf.random.uniform((D, H)))  # weights
w2 = tf.Variable(tf.random.uniform((H, D)))  # weights

optimizer = tf.optimizers.SGD(1e-6)

for t in range(50):
  with tf.GradientTape() as tape:
    h = tf.maximum(tf.matmul(x, w1), 0)
    y_pred = tf.matmul(h, w2)
    diff = y_pred - y
    loss = tf.losses.MeanSquaredError()(y_pred, y)
  gradients = tape.gradient(loss, [w1, w2])
  optimizer.apply_gradients(zip(gradients, [w1, w2]))
```

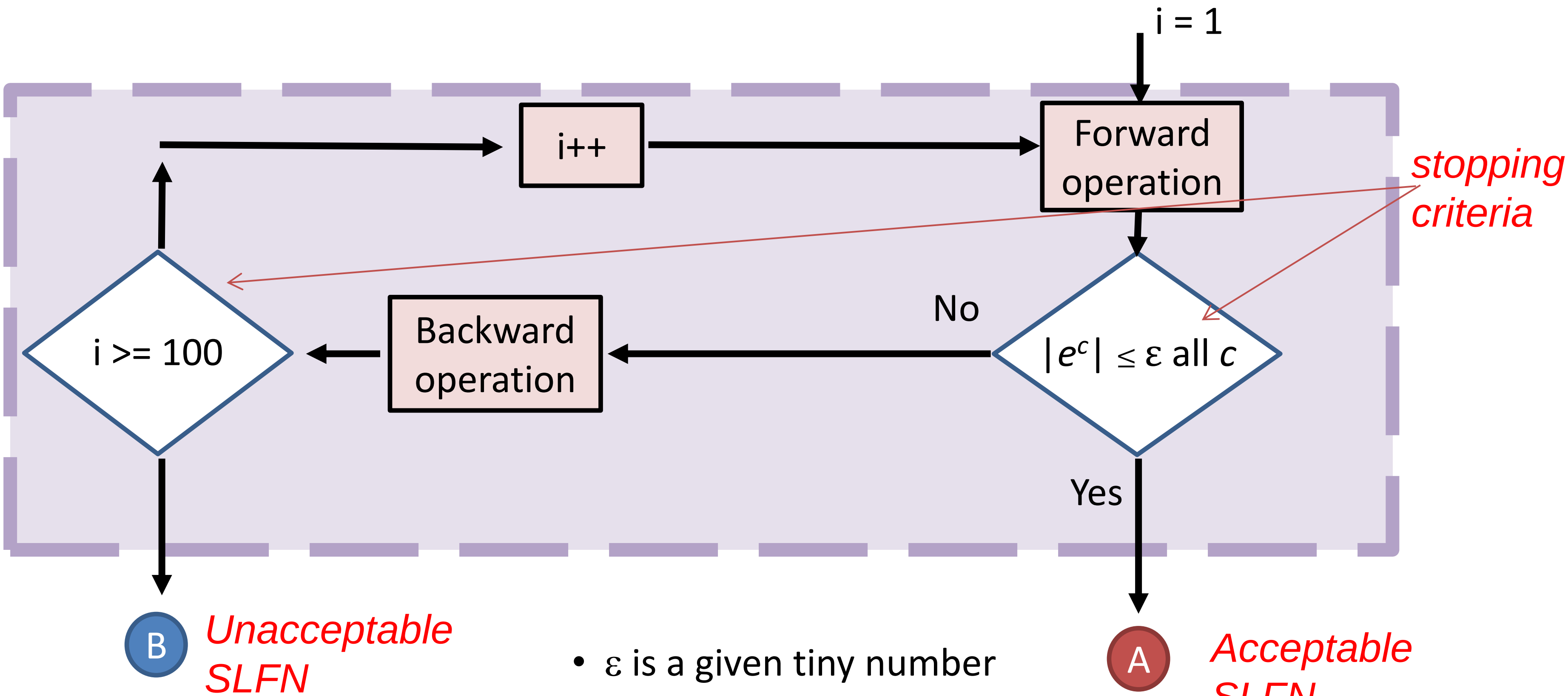
Keras: High-Level Wrapper

Keras is a layer on top of TensorFlow, makes common things easy to do



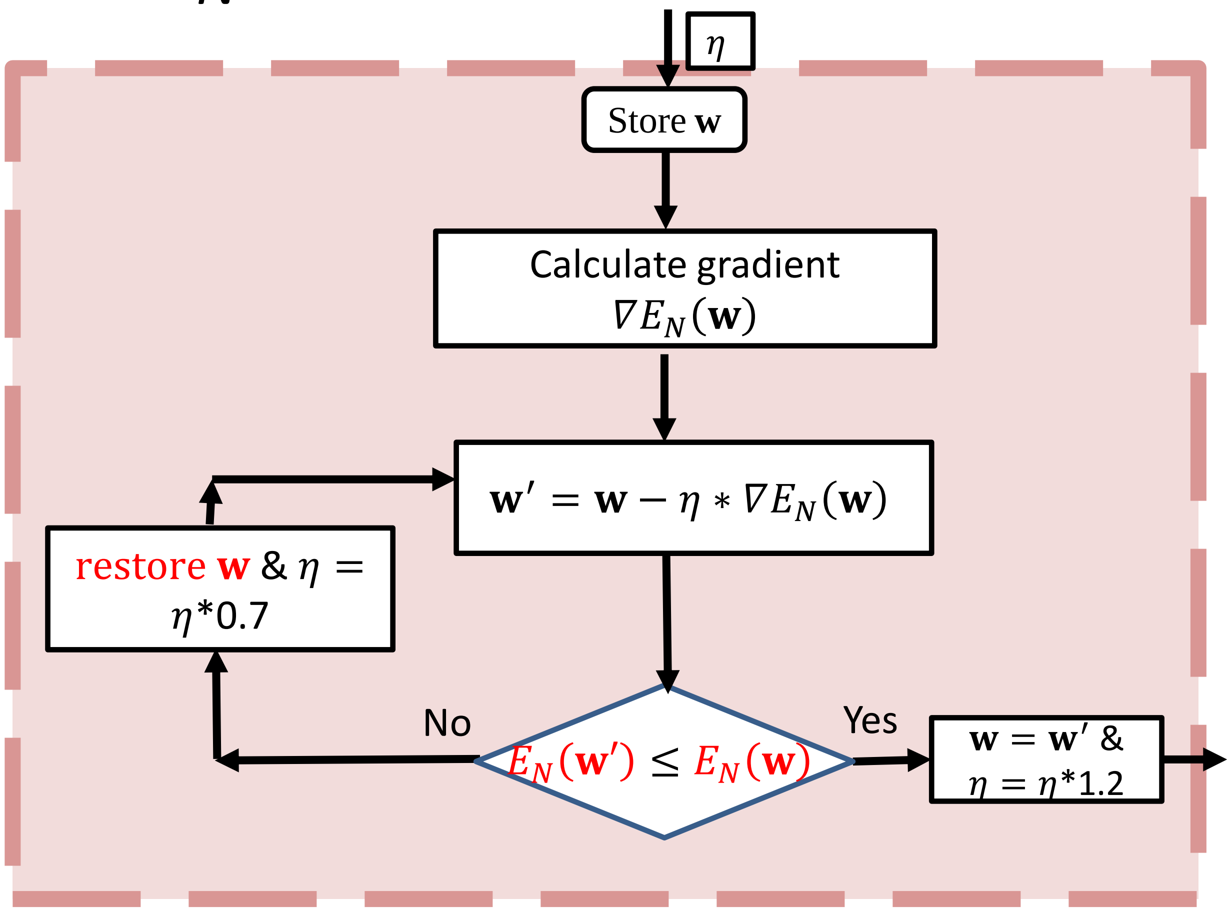

```
N, D, H = 64, 1000, 100

x = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
y = tf.convert_to_tensor(np.random.randn(N, D), np.float32)
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(H, input_shape=(D,),
                                activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(D))
optimizer = tf.optimizers.SGD(1e-1)

losses = []
for t in range(50):
  with tf.GradientTape() as tape:
    y_pred = model(x)
    loss = tf.losses.MeanSquaredError()(y_pred, y)
  gradients = tape.gradient(
      loss, model.trainable_variables)
  optimizer.apply_gradients(
      zip(gradients, model.trainable_variables))
```

The algorithm with an extra stopping criterion that indicates either an undesired SLFN or a desired SLFN



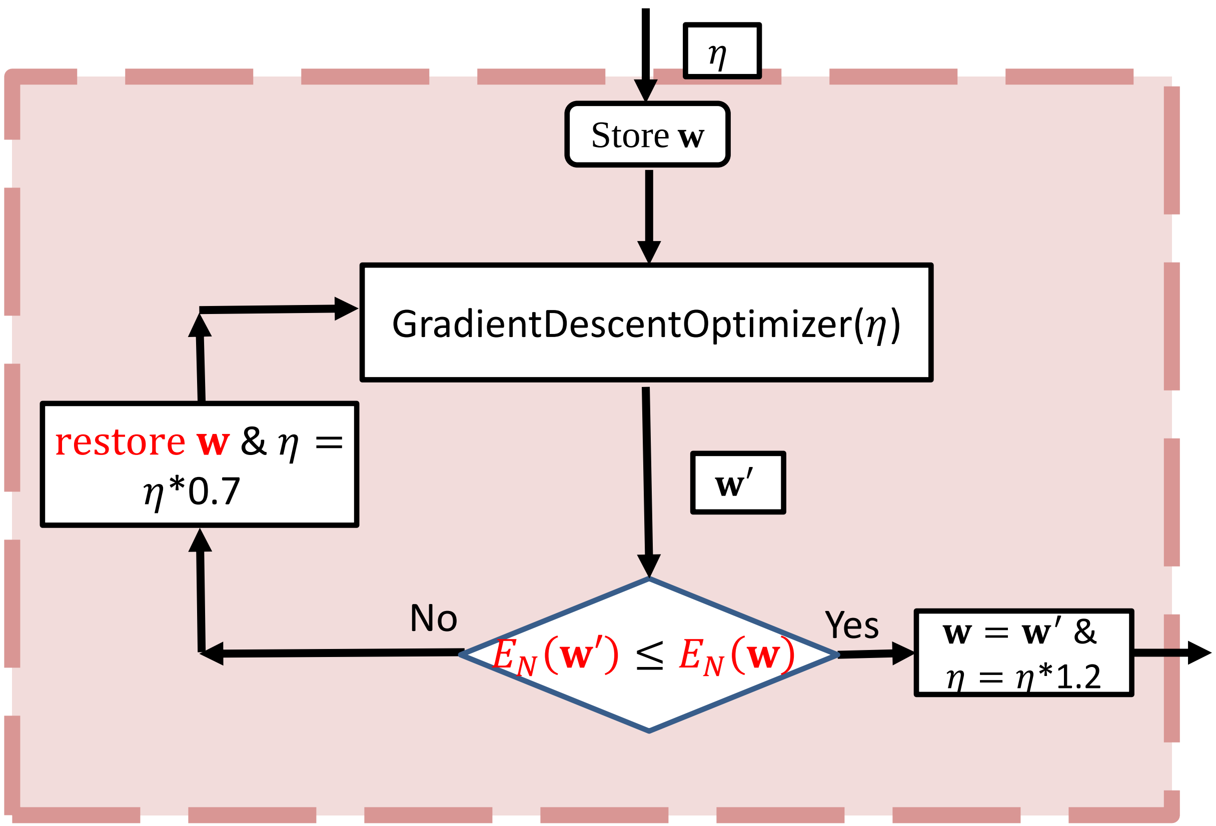
- ε is a given tiny number
- $e^c = f(\mathbf{x}^c, \mathbf{w}) - y^c$

The adaptable η arrangement in the backward operation module for guaranteeing the decrease of $E_N(\mathbf{w})$

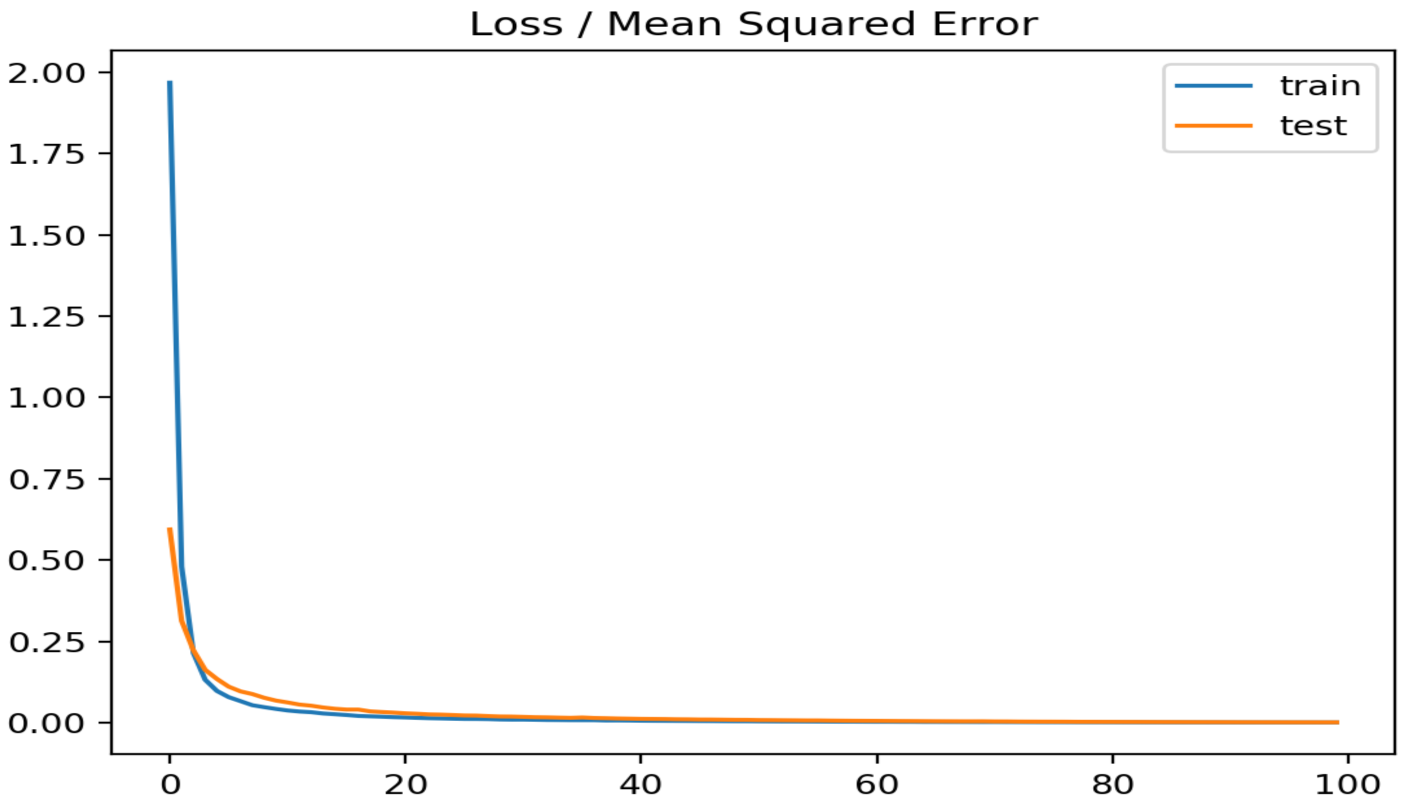


Where we are now…

The adaptable η arrangement in the backward operation module with GradientDescentOptimizer



Adaptable learning rate

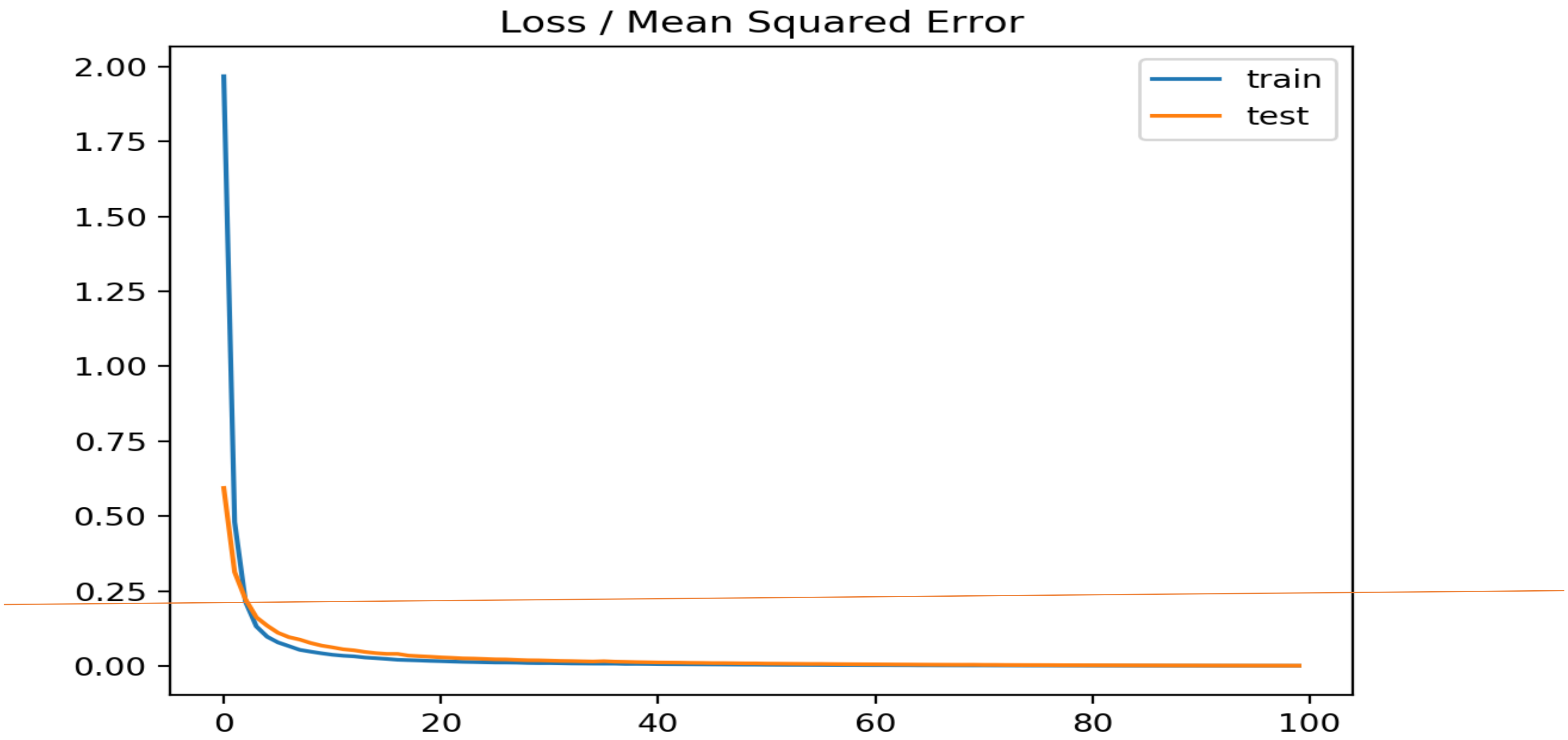


Stopping criteria and the learning goals for the learning

The learning process should stop when

1. Hit the epoch constraint (e.g., i >= 100)
2. ~~$E_N(\mathbf{w}) = 0$~~
3. ~~Obtain a tiny $E_N(\mathbf{w})$ value~~
4. $|f(\mathbf{x}^c, \mathbf{w}) - y^c| < \varepsilon \;\forall\, c \textit{ where } \varepsilon \textit{ is tiny}$

The learning goal



Extra stopping criteria (but not learning goals) for the learning

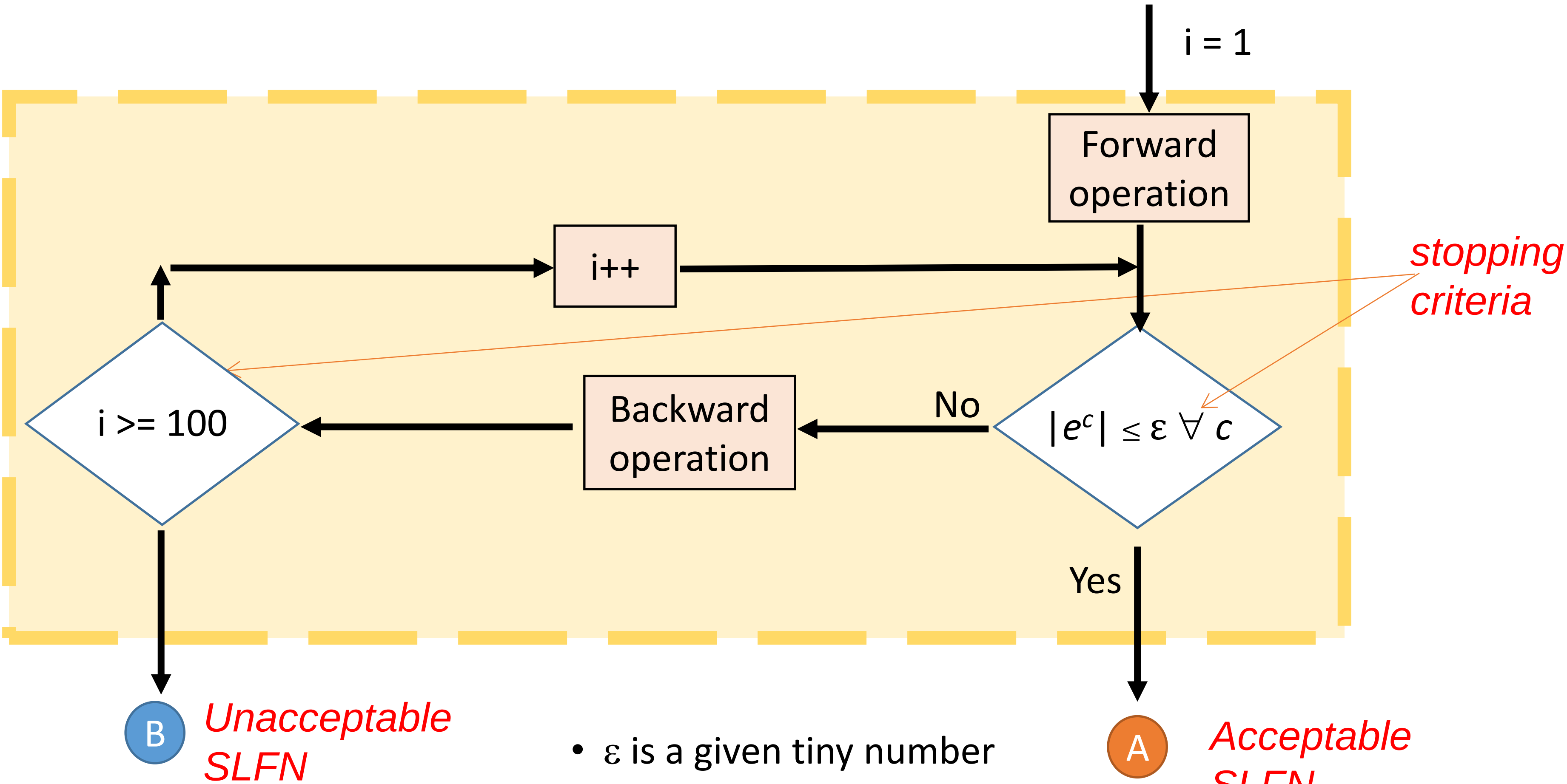
1. ~~The learning process should stop when~~ $\cancel{\|\nabla_{\mathbf{w}} E_N(\mathbf{w})\| = 0.}$
2. The learning process should stop when $\|\nabla_{\mathbf{w}} E_N(\mathbf{w})\|$ is tiny.
3. The learning process should stop when (adaptive) η (the learning rate) is tiny.

The undesired attractors:

a) the local optimum/the saddle point/the plateau

b) the global optimum of the defective network architecture

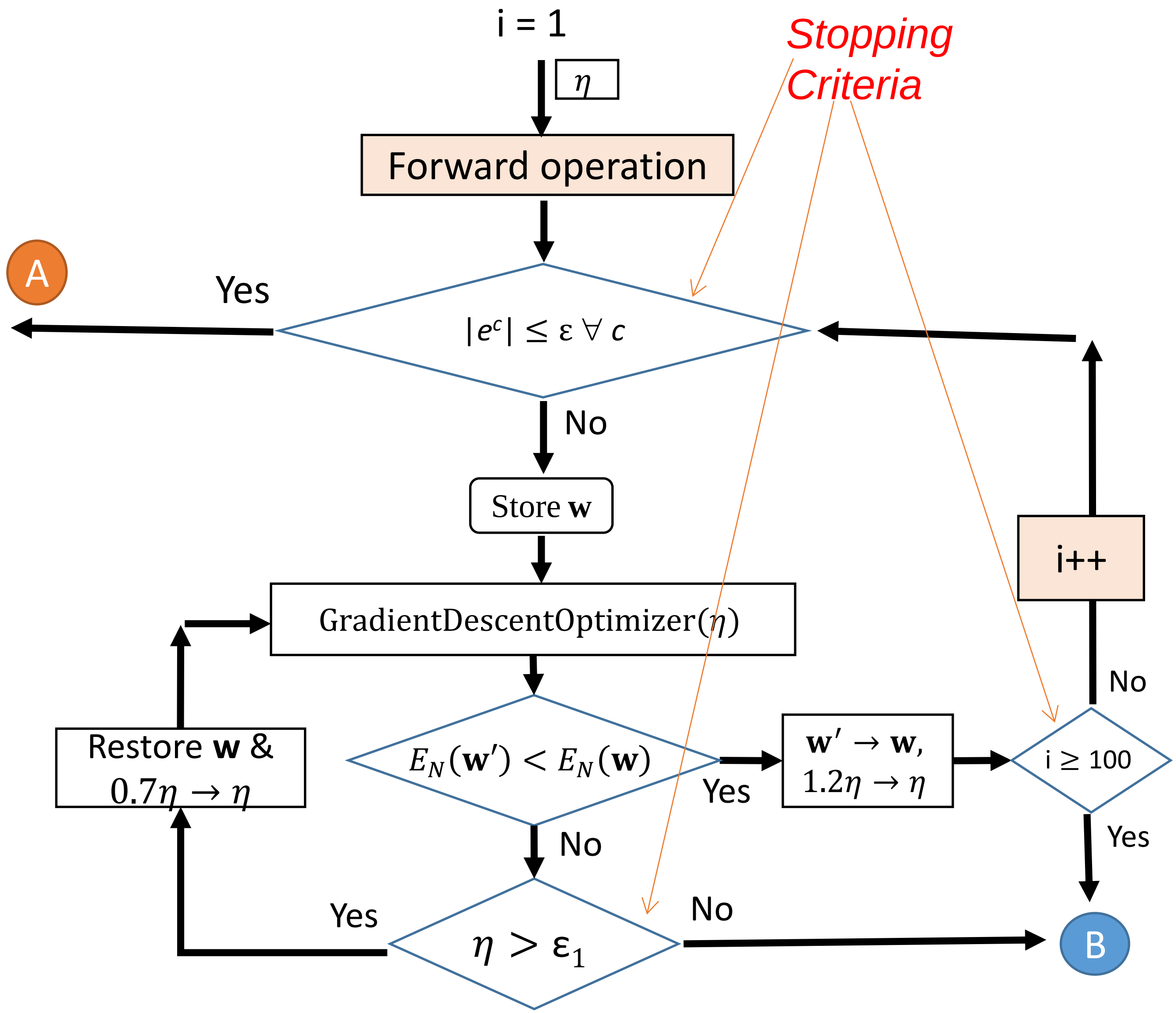
The algorithm with extra stopping criteria



- ε is a given tiny number
- $e^c = f(\mathbf{x}^c, \mathbf{w}) - y^c$

Where we are now...

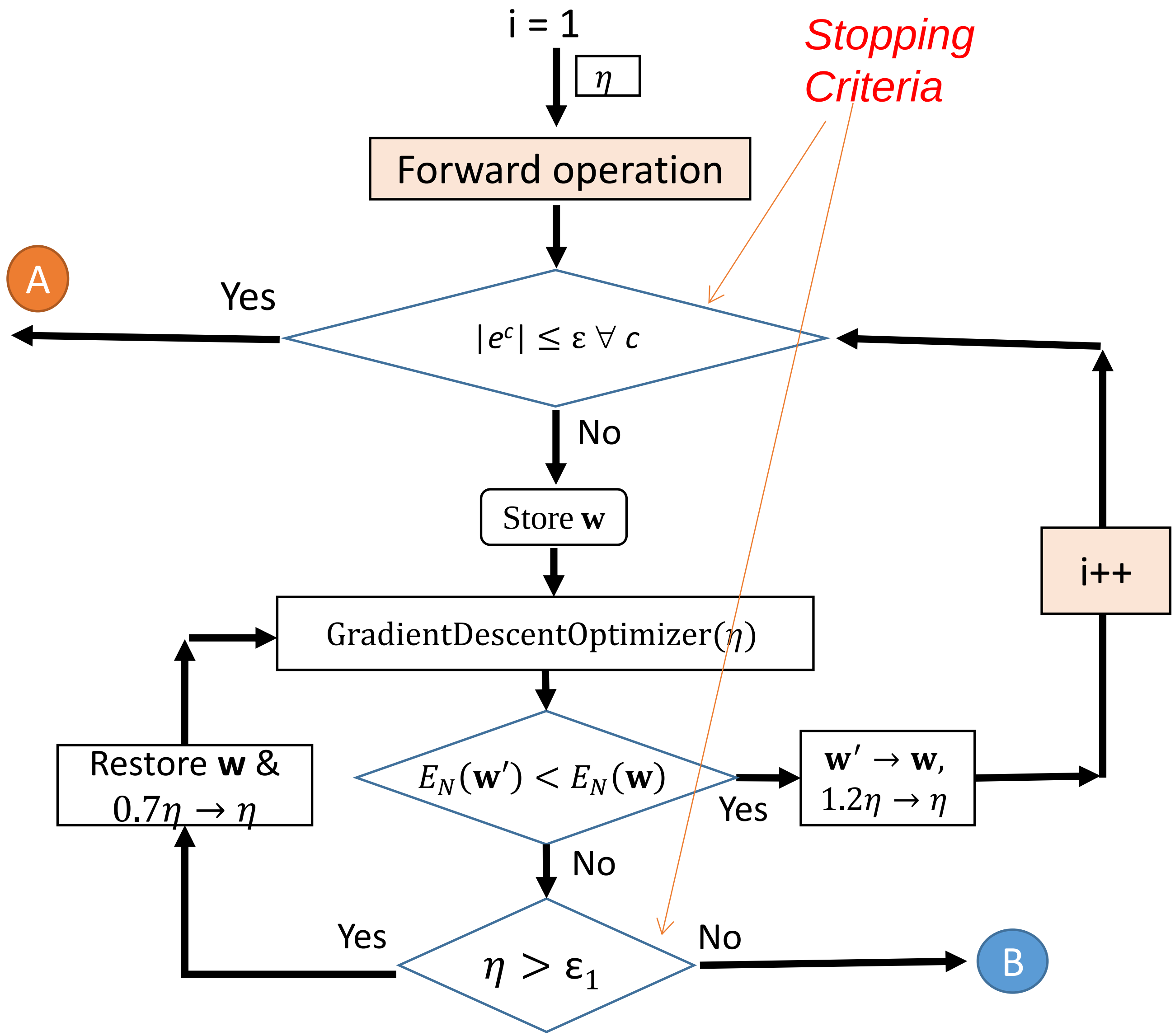
the algorithm for SLFNs with extra stopping criteria that indicate whether the final result is either an undesired SLFN or a desired SLFN.



- ε and ε_1 are given tiny numbers
- $e^c = f(\mathbf{x}^c, \mathbf{w}) - y^c$

Where we are now...

the algorithm for SLFNs with extra stopping criteria that indicate whether the final result is either an undesired SLFN or a desired SLFN.

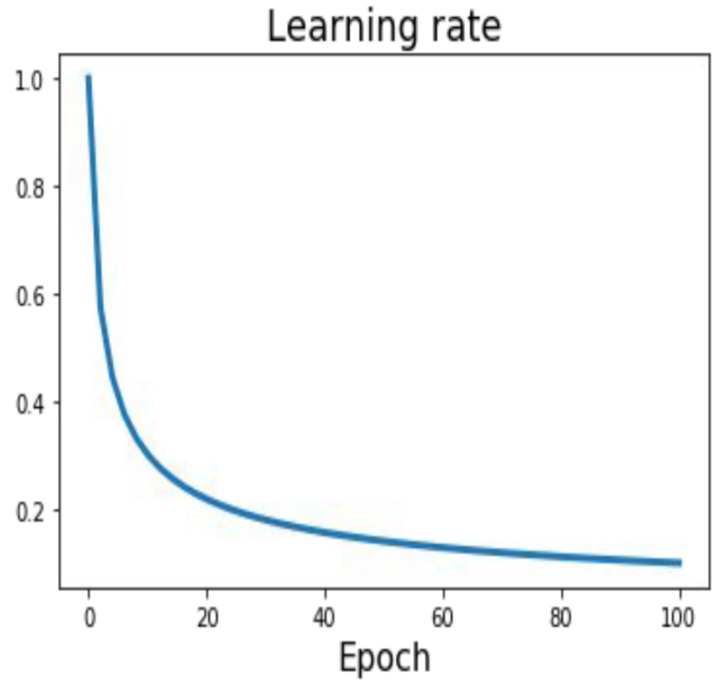


- ε and ε_1 are given tiny numbers
- $e^c = f(\mathbf{x}^c, \mathbf{w}) - y^c$

In the backward operation module

The Adaptable
Learning Rate Arrangement vs
The Learning Rate Decay

Learning Rate Decay



Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

Cosine: $\alpha_t = \frac{1}{2}\alpha_0 \left(1 + \cos(t\pi/T)\right)$

Linear: $\alpha_t = \alpha_0(1 - t/T)$

Inverse sqrt: $\alpha_t = \alpha_0/\sqrt{t}$

α_0 : Initial learning rate
α_t : Learning rate at epoch t
T : Total number of epochs

Vaswani et al, "Attention is all you need", NIPS 2017

New algorithm & Coding

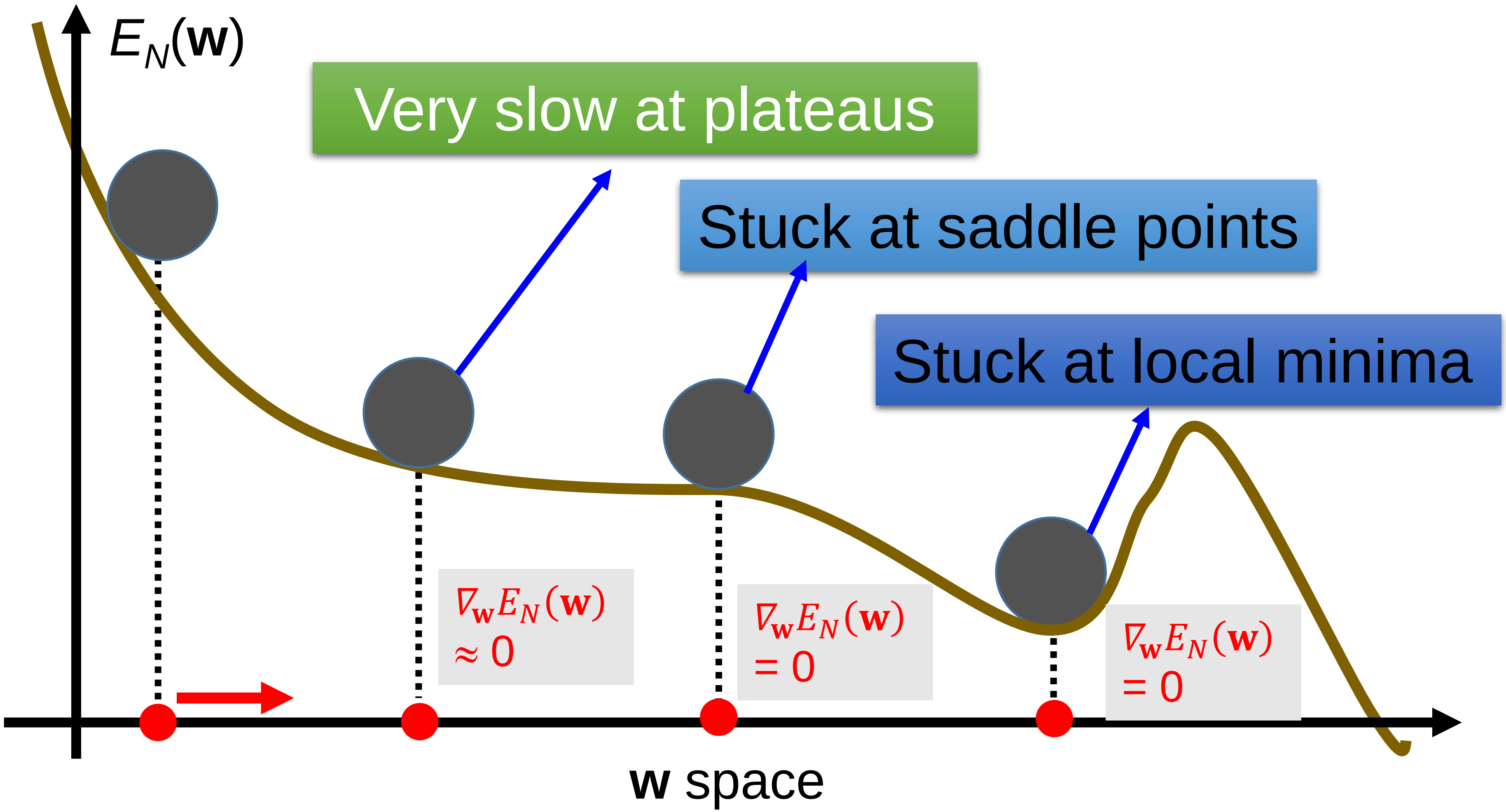
- Not merely double check the correctness of codes
- New AI algorithm → new learning process ← Double check the learning process (ALWAYS!!!)
- Simple checks: (1) whether the evolution of $E(\mathbf{w})$ values is reasonable? (2) whether the tuning of $\mathbf{w}$ is reasonable? (3) whether the evolution of $f(\mathbf{x}^c, \mathbf{w})$ values all c is reasonable?
- Complicated checks: whether the learning process is reasonable?

Not the performance, which is related with the inferencing.

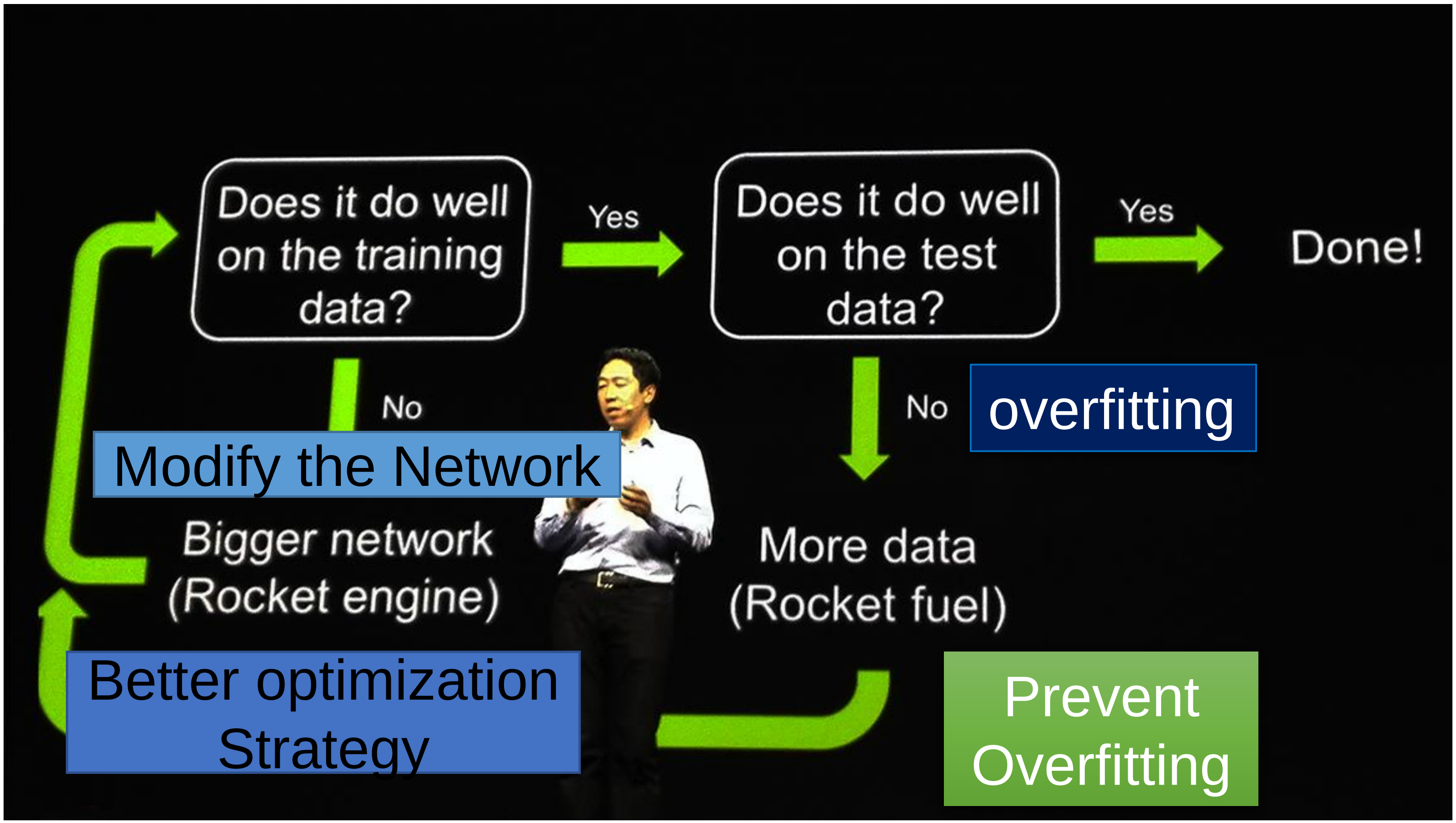
Performance of AI Applications

- How do AI professionals or high-rank managers evaluate the performance of the AI applications? ← effectiveness & efficiency
- However, there are learning dilemma and overfitting in front of the discussion of effectiveness & efficiency.

You need to deal with undesired attractors. Not only for the purposes of learning, but inferencing.

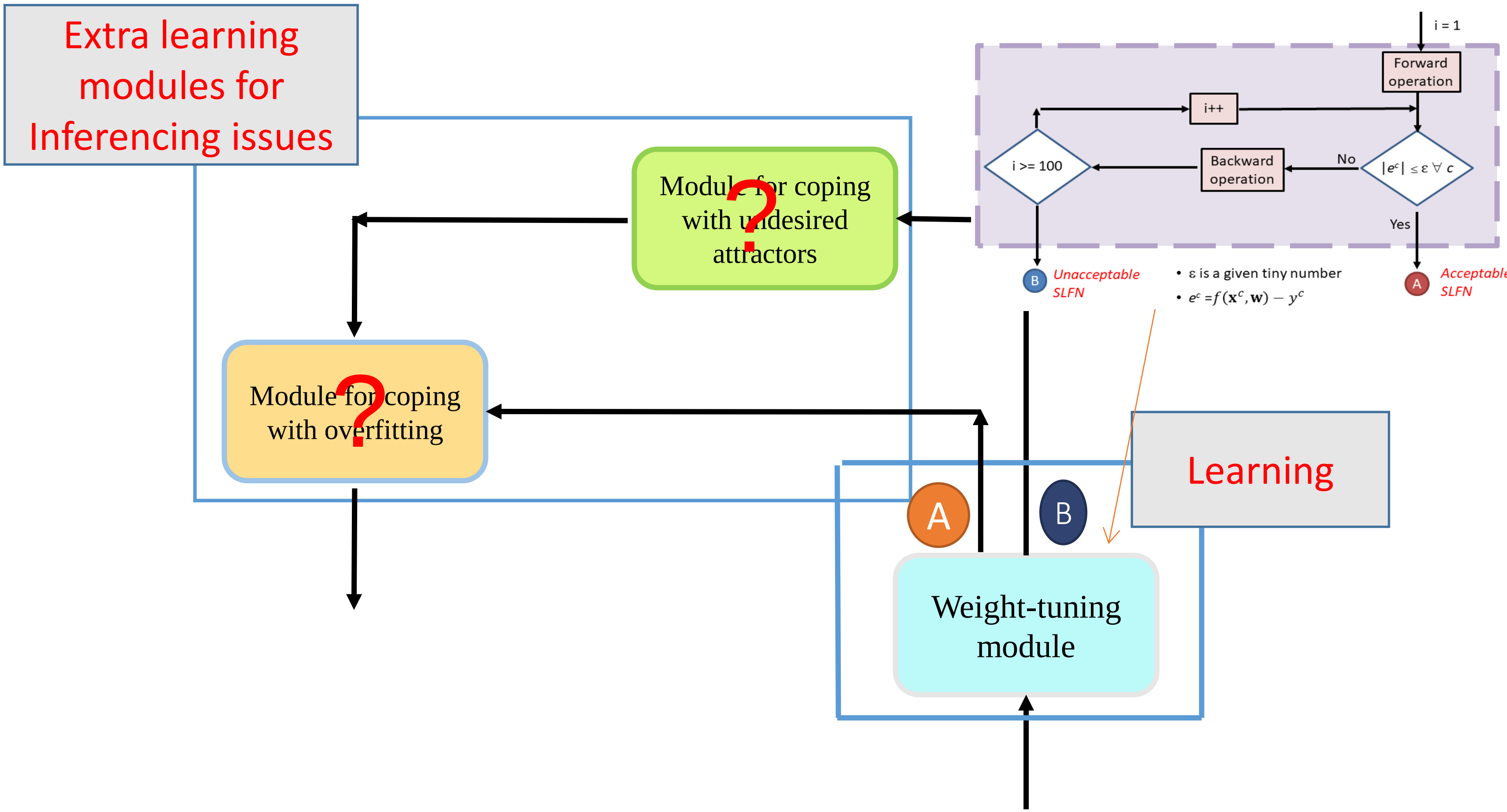


Recipe for Deep Learning



http://www.gizmodo.com.au/2015/04/the-basic-recipe-for-machine-learning-explained-in-a-single-powerpoint-slide/

Inferencing Issues



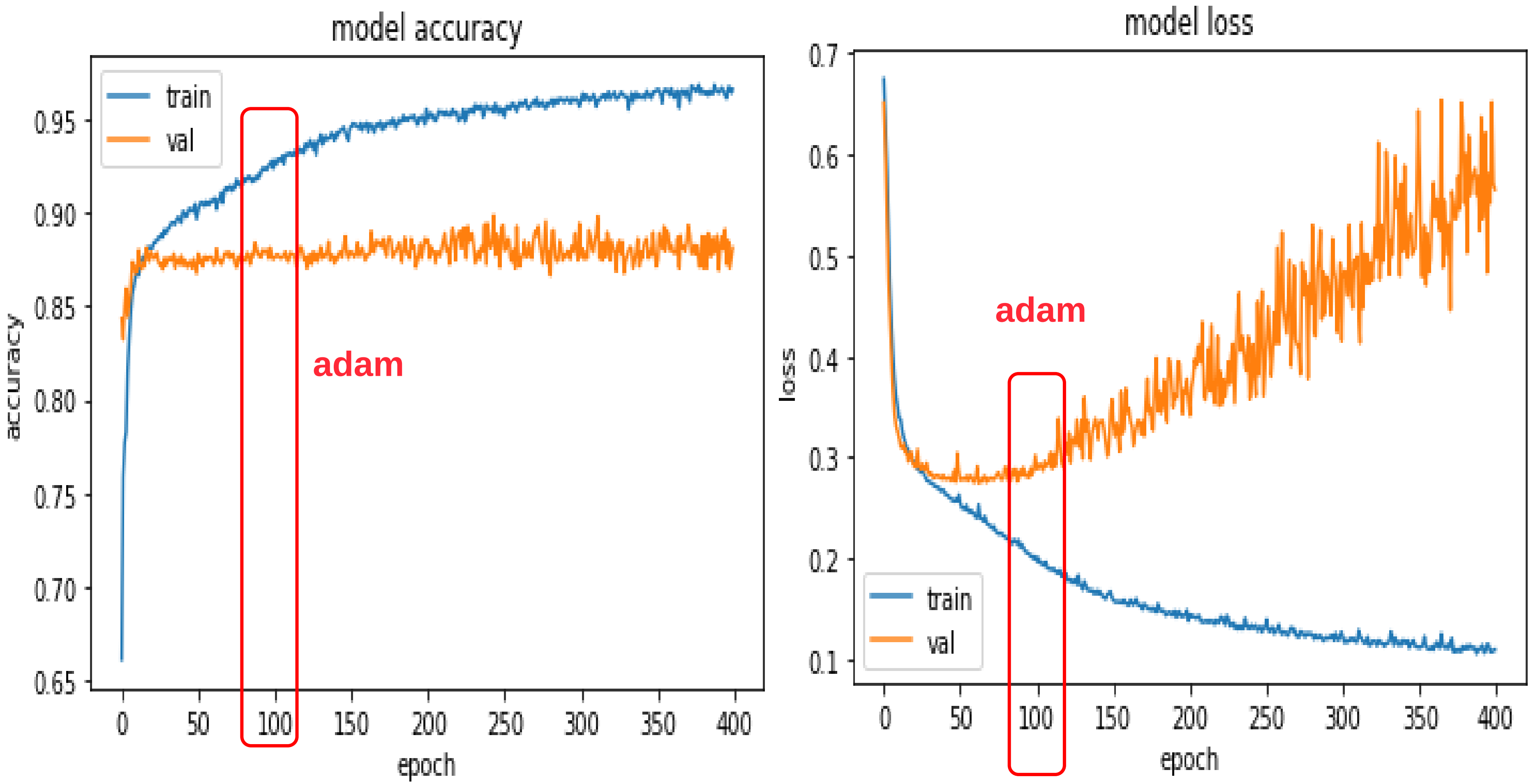
Generalization

- Learned hypothesis may **fit** the **training data** very well, even **noises** (outliers in the training data), but fail to **generalize** to **new examples (test data**)
- In machine learning and statistical learning theory, **generalization error** (also known as the **out-of-sample error**) is a measure of how accurately an algorithm is able to predict outcome values for previously unseen data.

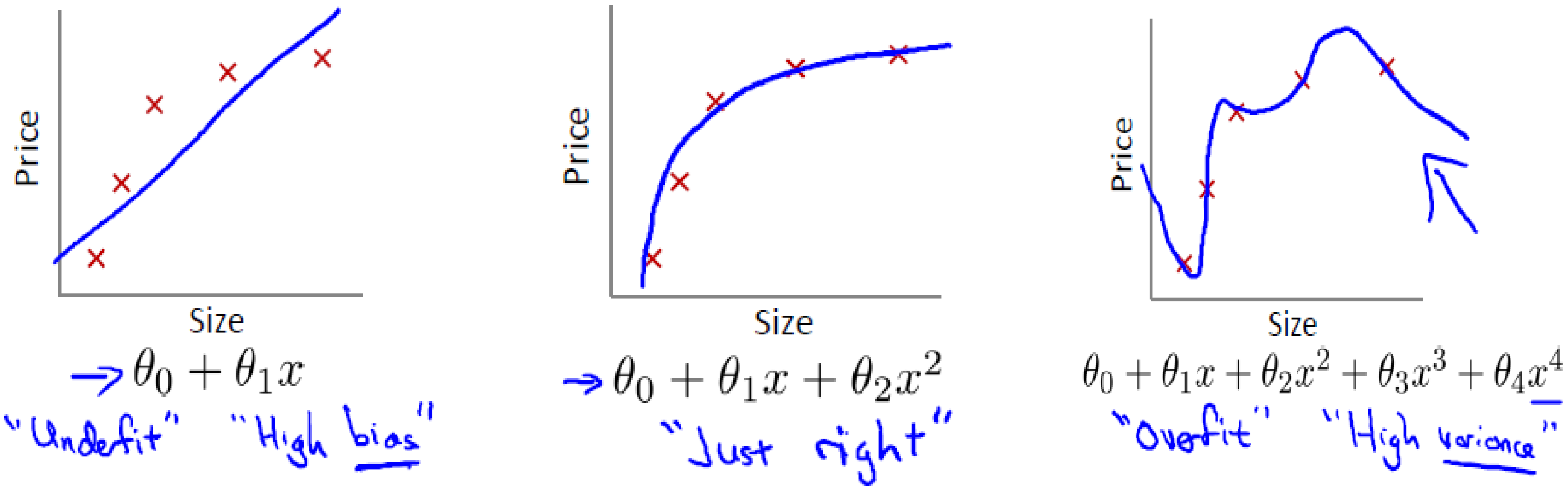
Learning curves

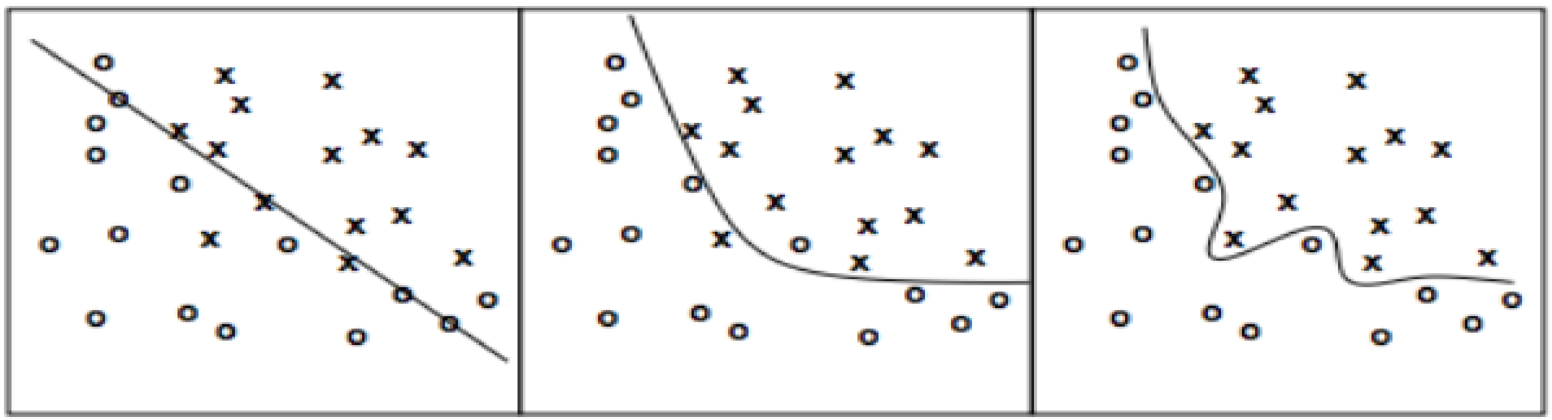
- Because learning algorithms are evaluated on finite samples, the evaluation of a learning algorithm may be sensitive to sampling error. As a result, measurements of prediction error on the current data may not provide much information about predictive ability on new data.
- The performance of a learning algorithm is measured by plots of the generalization error values through the learning process, which are called learning curves.
- Generalization error can be minimized by avoiding overfitting in the learning process.

Learning curve and overfitting



overfitting



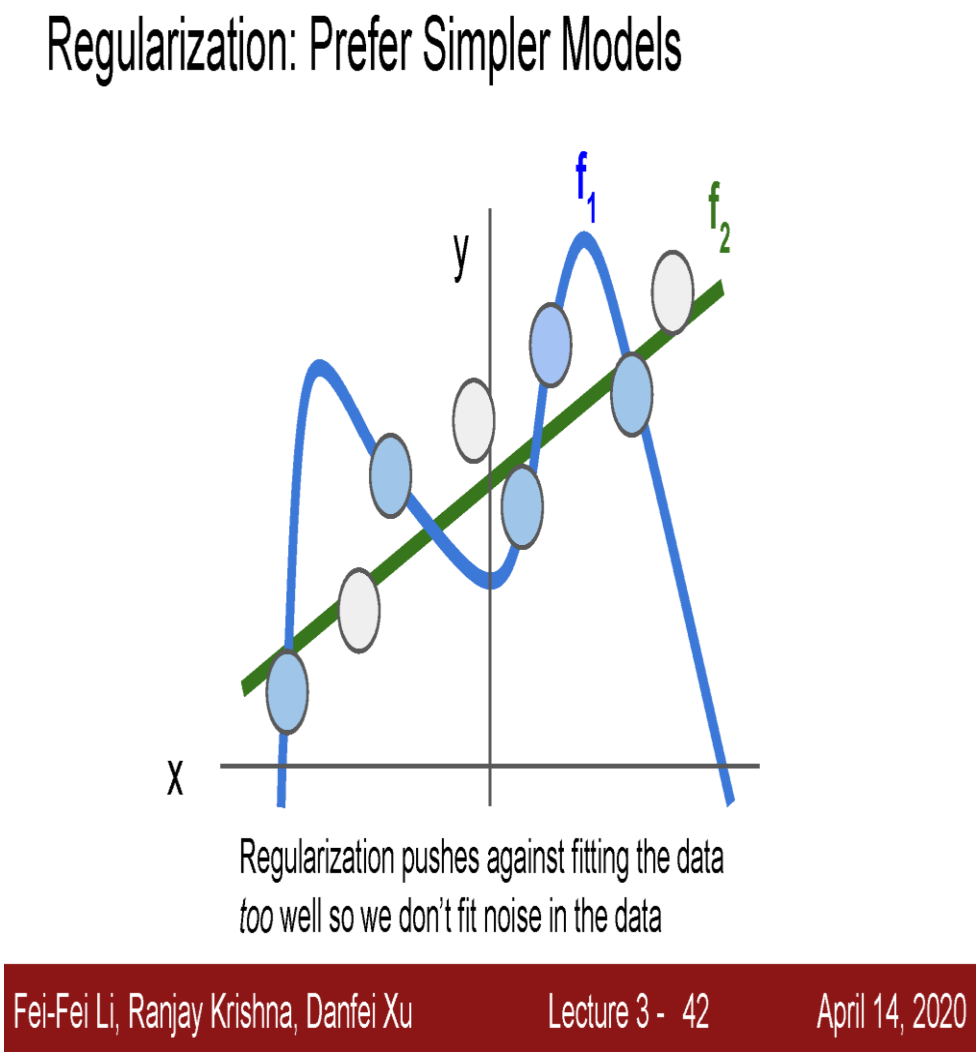


inadequate good compromise over-fitting

http://wiki.bethanycrane.com/overfitting-of-data

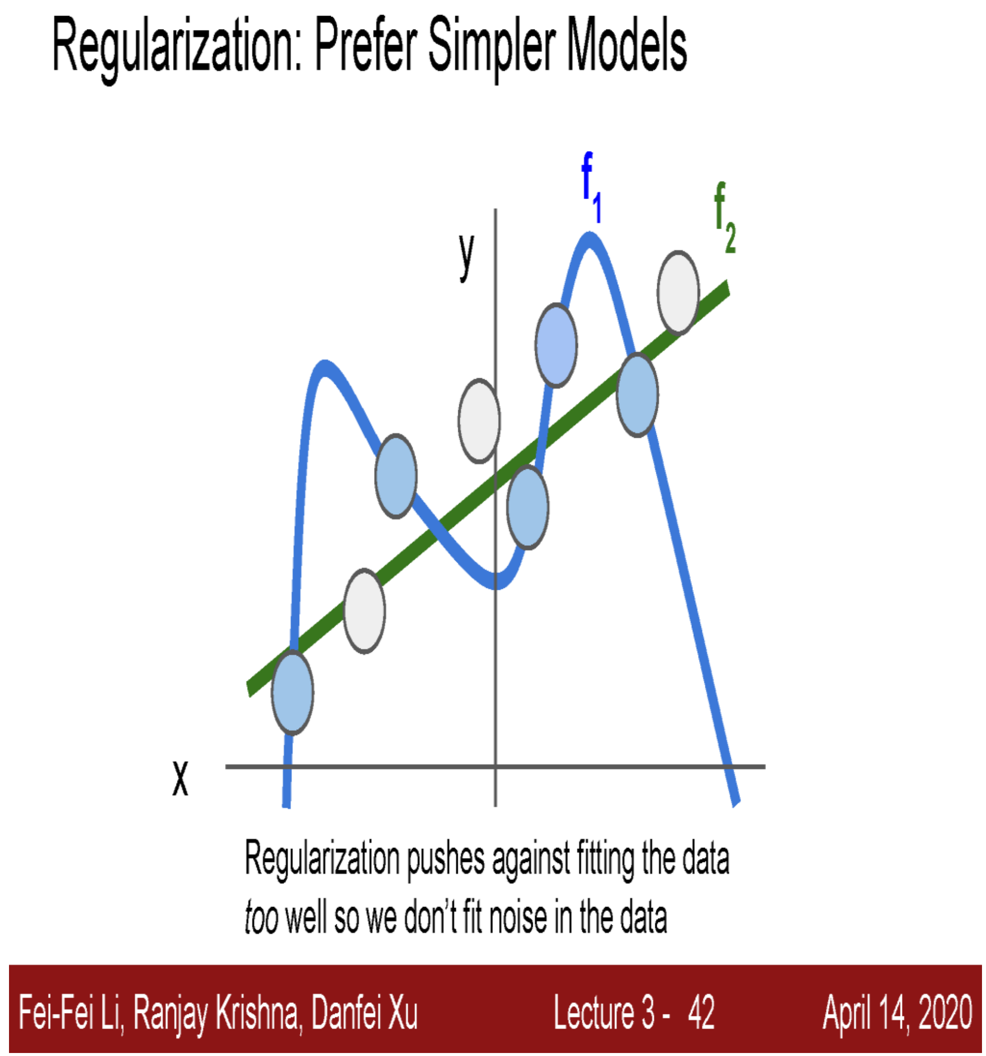
Overfitting

In statistics, **overfitting** is "the production of an analysis that corresponds too closely or exactly to a particular set of data, and may therefore fail to fit additional data or predict future observations reliably."



Overfitting

An **over-fitted model** is a model that contains more parameters than can be justified by the data.



Overfitting due to big weights

- To penalize big weights, there is a regularization term in the loss function.
- The loss function:

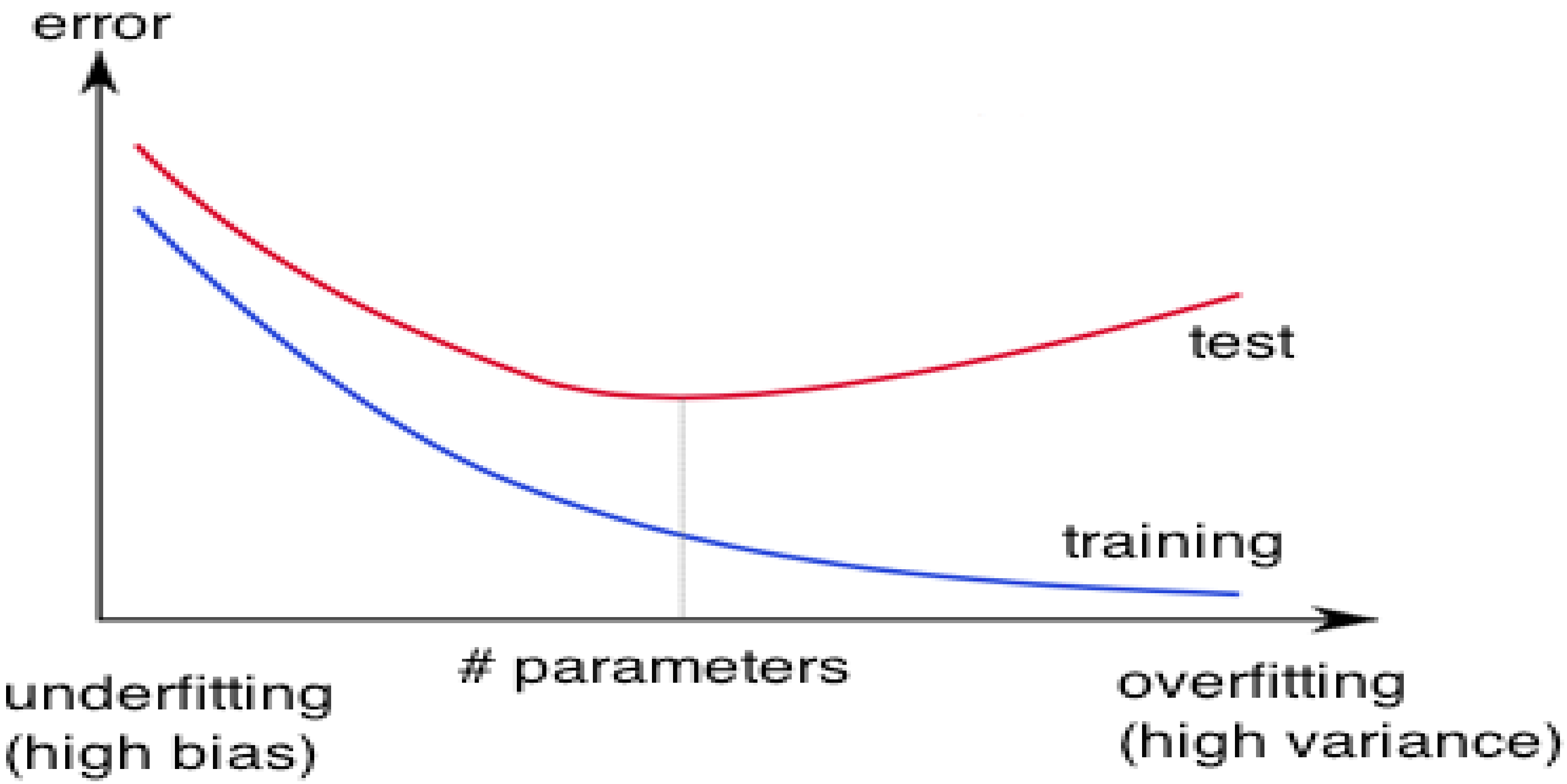
> - decay term: tiny λ
> - Regularization term: arbitrary λ

$$E_N(\mathbf{w}) \equiv \frac{1}{N}\sum_{c=1}^{N}(f(\mathbf{x}^c, \mathbf{w}) - y^c)^2 + \lambda\|\mathbf{w}\|^2$$

- The weight decay coefficient λ determines how dominant the regularization is during gradient computation
- Big weight decay coefficient → big penalty for big weights

- The above is the L2 regularization term
- L1 regularization: $\lambda|\mathbf{w}|$
- Elastic net: L1 + L2

Overfitting due to too many hidden nodes



https://www.neuraldesigner.com/images/learning/selection_error.svg

Regularization

$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)} + \underbrace{\lambda R(W)}$$

Data loss: Model predictions should match training data

Regularization: Prevent the model from doing *too* well on training data

Occam's Razar: Among multiple competing hypotheses, the simplest is the best,
William of Ockham 1285-1347

Regularization

λ = regularization strength (hyperparameter)

$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)}_{\text{Data loss}} + \underbrace{\lambda R(W)}_{\text{Regularization}}$$

Data loss: Model predictions should match training data

Regularization: Prevent the model from doing *too* well on training data

Why regularize?
- Express preferences over weights
- Make the model *simple* so it works on test data
- Improve optimization by adding curvature

Regularization

λ = regularization strength (hyperparameter)

$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)} + \underbrace{\lambda R(W)}$$

Data loss: Model predictions should match training data

Regularization: Prevent the model from doing *too* well on training data

Simple examples

L2 regularization: $R(W) = \sum_k \sum_l W_{k,l}^2$

L1 regularization: $R(W) = \sum_k \sum_l |W_{k,l}|$

Elastic net (L1 + L2): $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

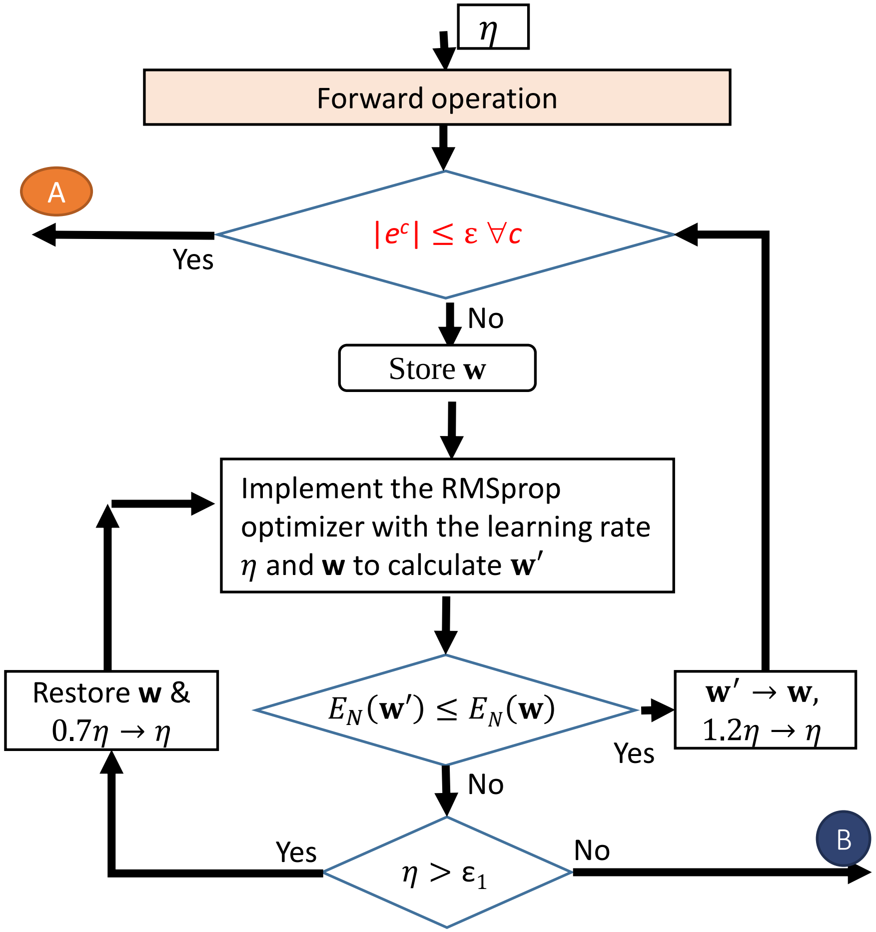
More complex:

Dropout

Batch normalization

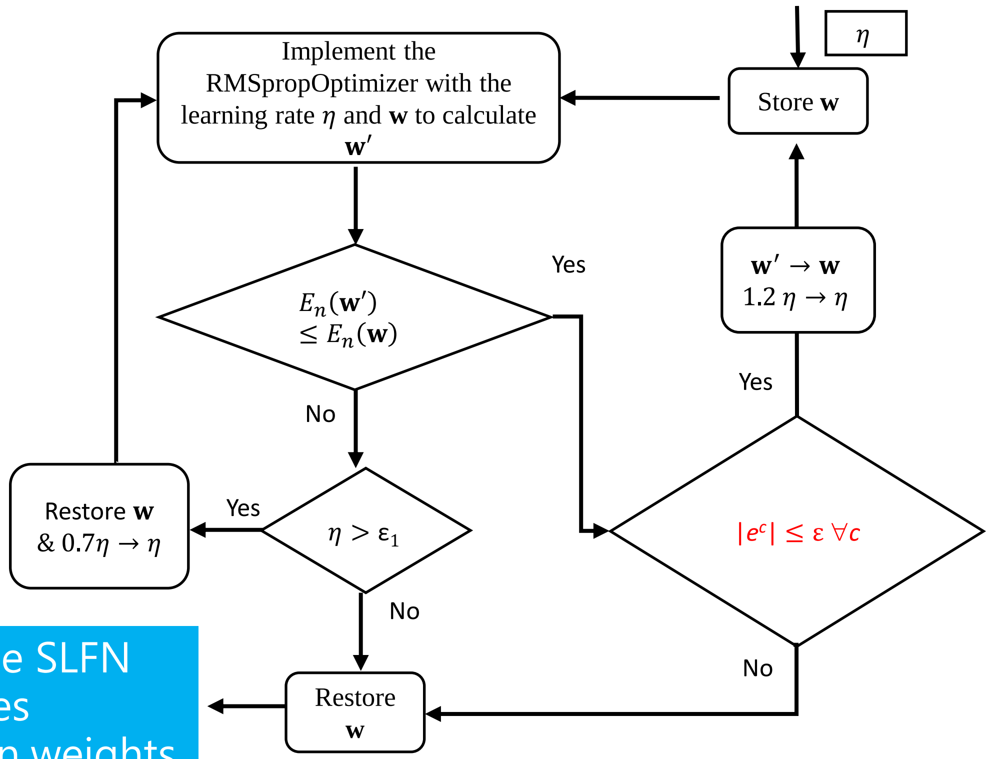
Stochastic depth, fractional pooling, etc

The weight-tuning module for learning



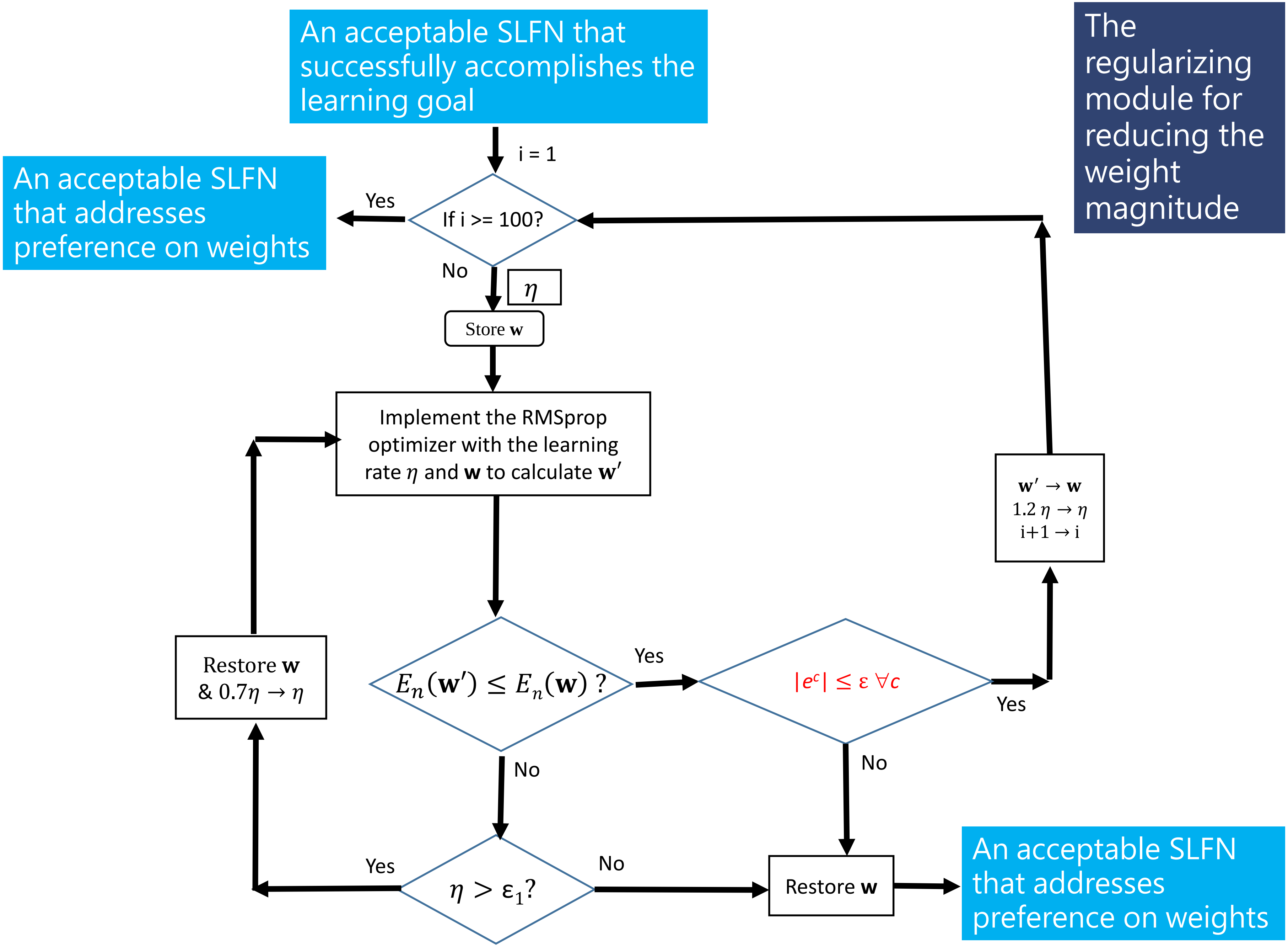
The regularizing module for reducing the weight magnitude

An acceptable SLFN that successfully accomplishes the learning goal



An acceptable SLFN that addresses preference on weights

The regularizing module for reducing the weight magnitude



Homework #4

Write down the code of the regularizing module that implements minimizing $E_N(\mathbf{w})$ to reduce the magnitude of $\mathbf{w}$, while keeping the learning goal satisfied.