Coding, learning goals and stopping criteria of learning algorithms

國立政治大學 資訊管理學系
蔡瑞煌 特聘教授

Codes for 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

Use this! It will make your life easier

```
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

Modules can contain weights or other modules

You can define your own Modules using autograd!

```
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()
```

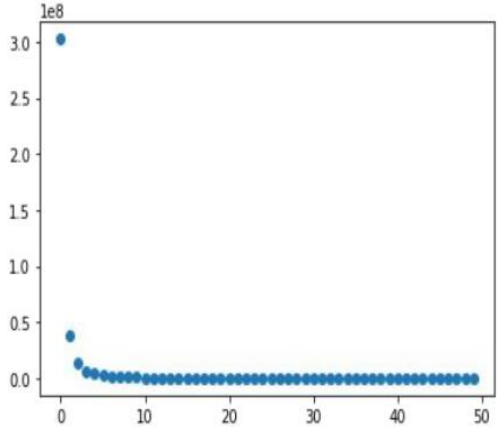
Where we are now...

TensorFlow: Neural Net

```
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])
```



Train the network: Run the training step over and over, use gradient to update weights

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

(Used to be third-party, now merged into TensorFlow)

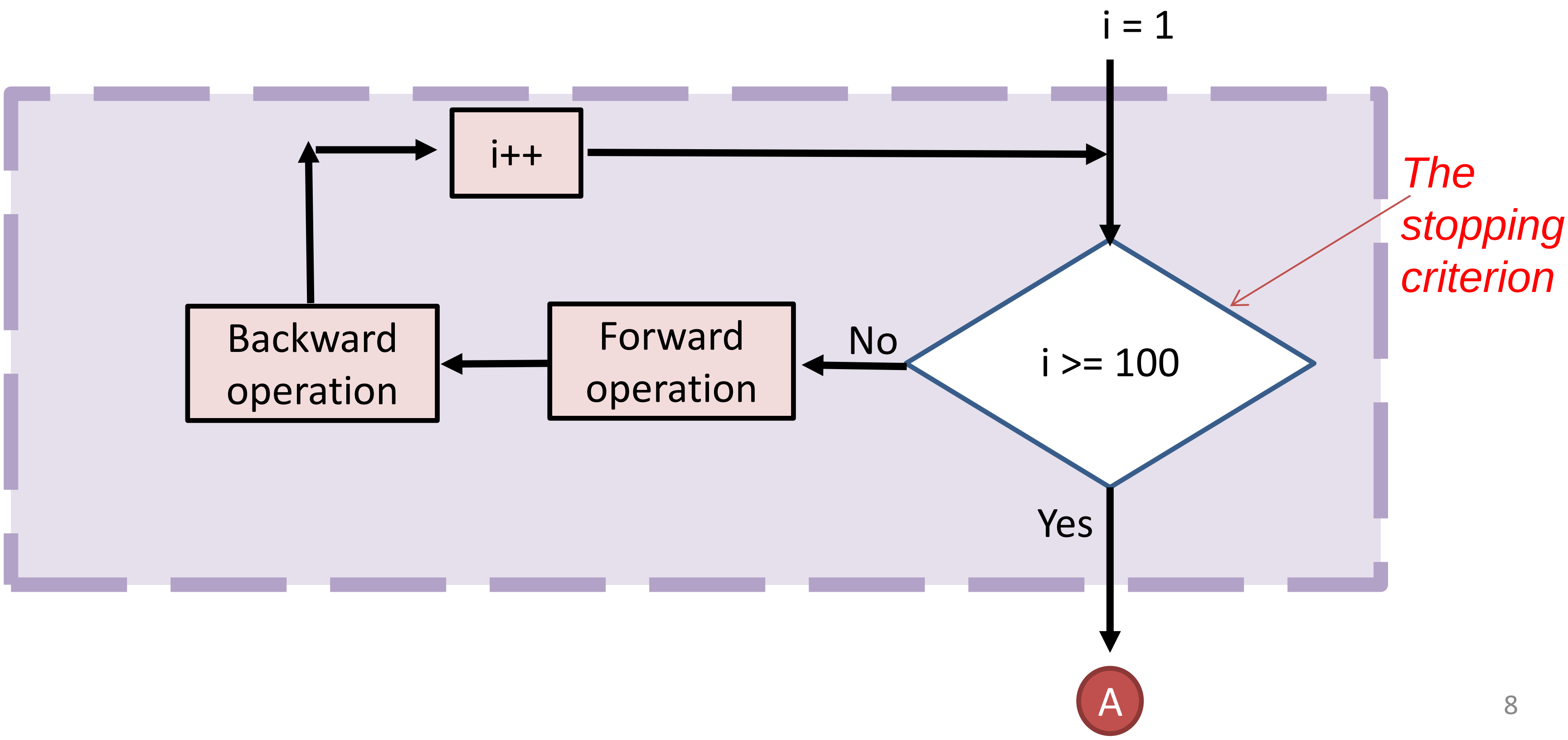
```
N, D, H = 64, 1000, 100

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))
```

Where we are now...

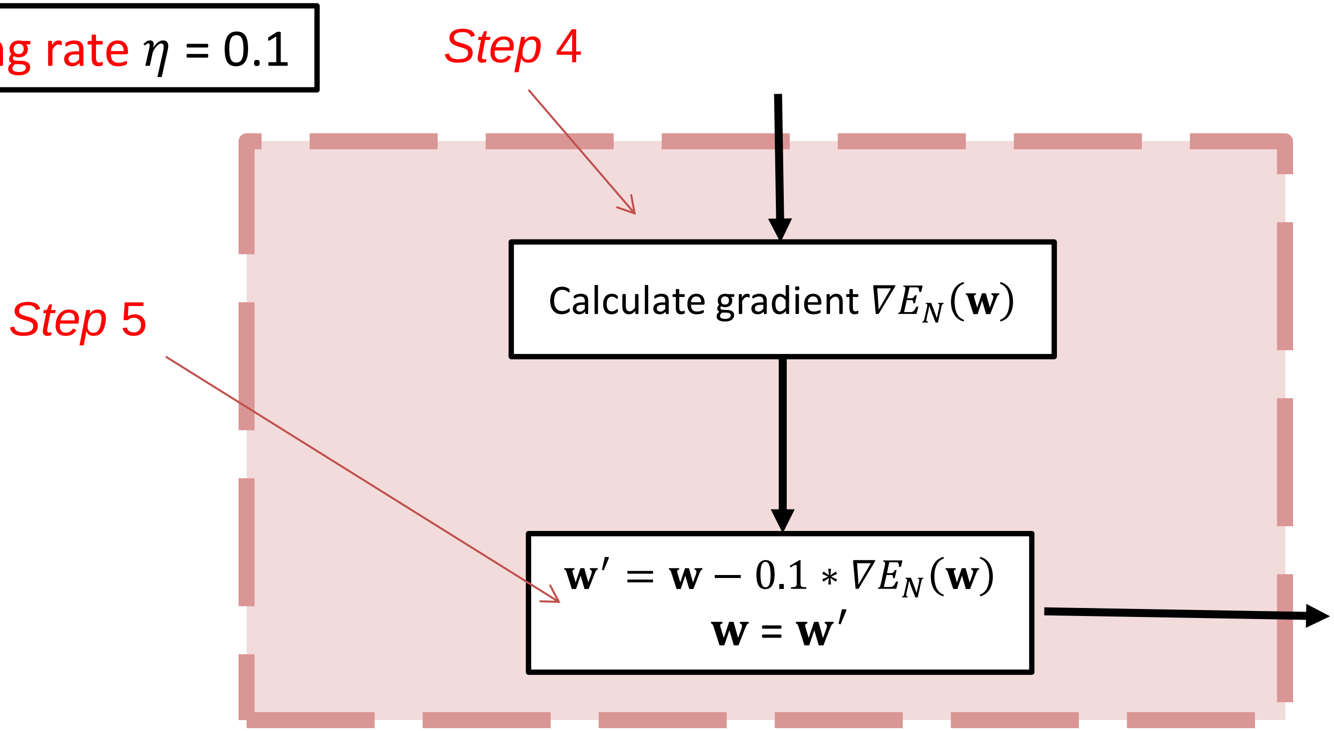
The learning algorithm



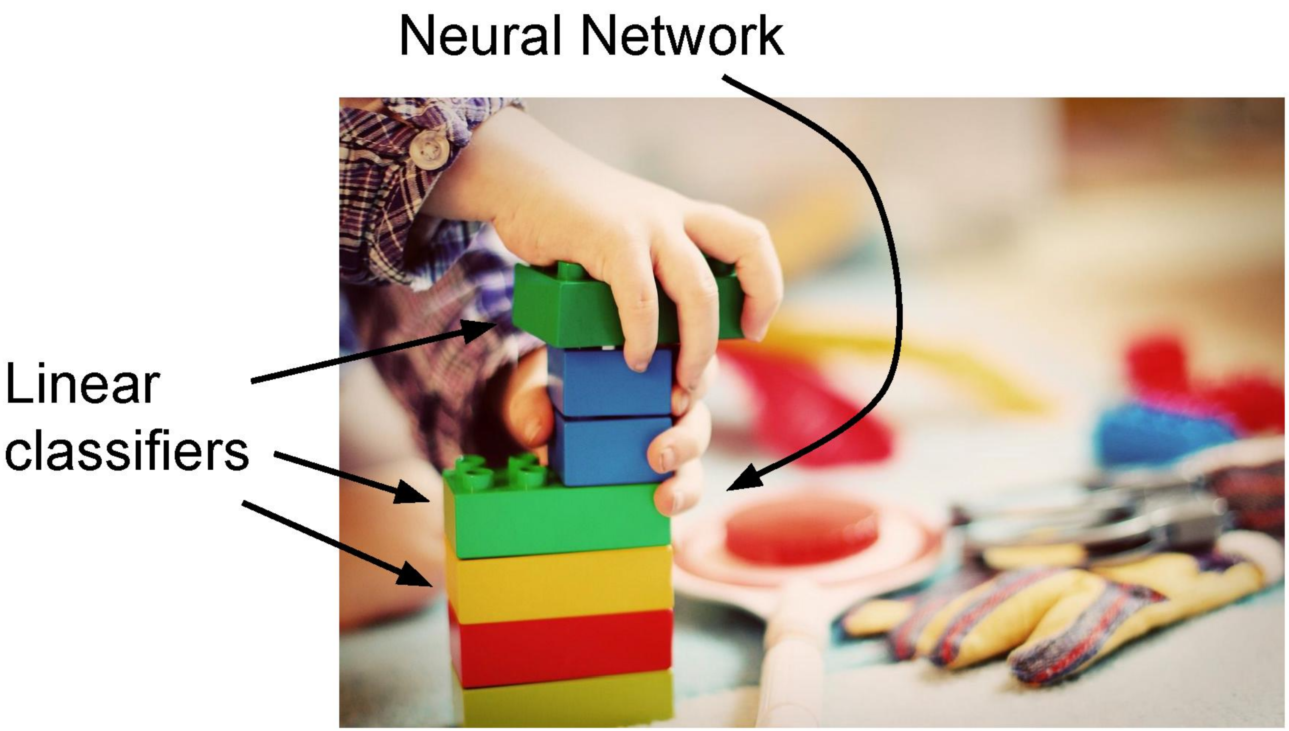
The Backward operation module

- Calculate the gradient and the adjustment of **w**

Learning rate η = 0.1



Developing a new AI system is like playing with Lego – lots of (pre-built or self-built) modules



idea/concept →

learning algorithm →

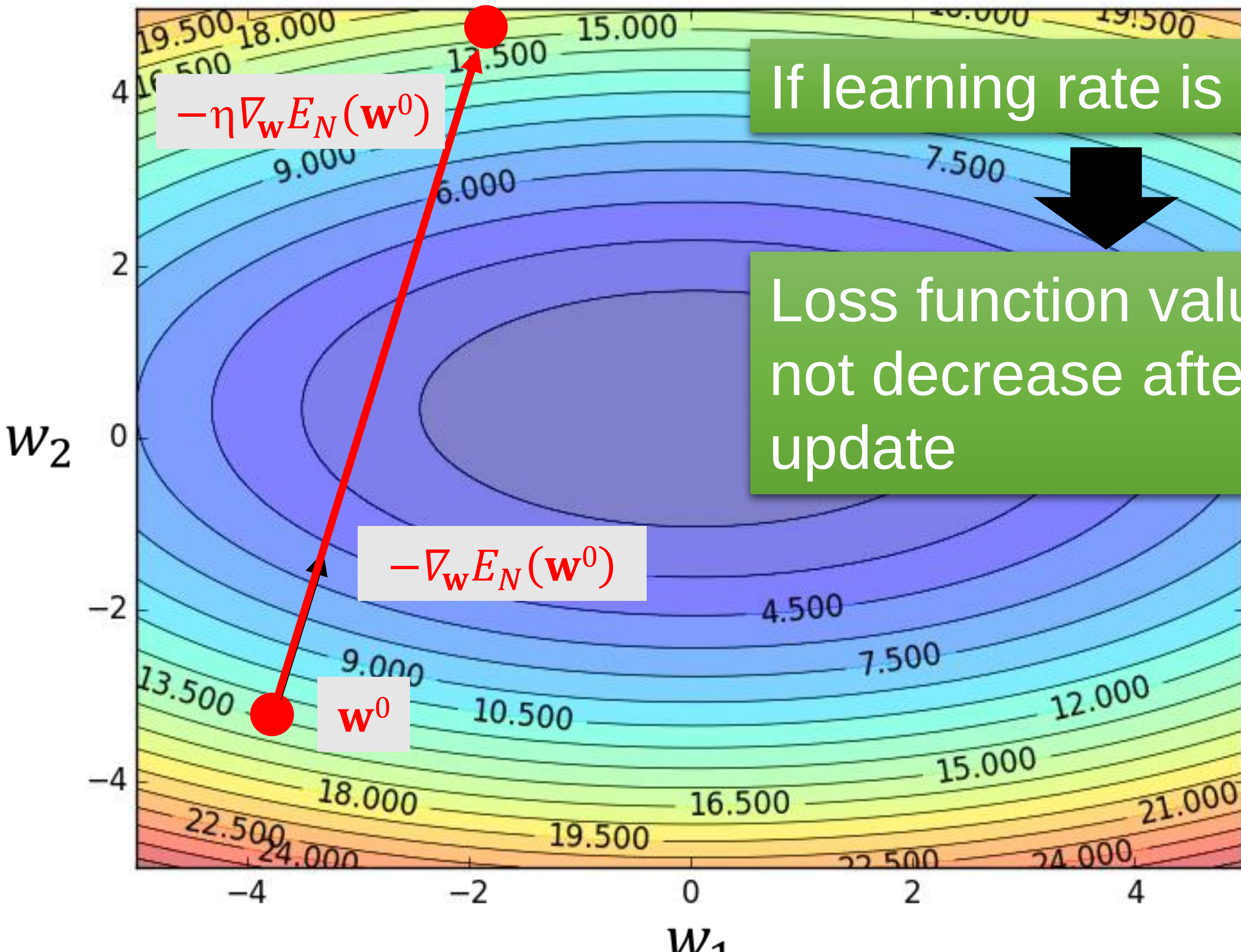
codes →

intelligent systems

The purpose of adjusting **w** and the learning rate

- What are the purposes of $\nabla_{\mathbf{w}} E_N(\mathbf{w})$ and $\Delta\mathbf{w}$?
- Can constantly reducing the $E_N(\mathbf{w})$ value be achieved via the fixed learning rate?

Learning Rate



If learning rate is too large

Loss function value may not decrease after each update

Learning Rate

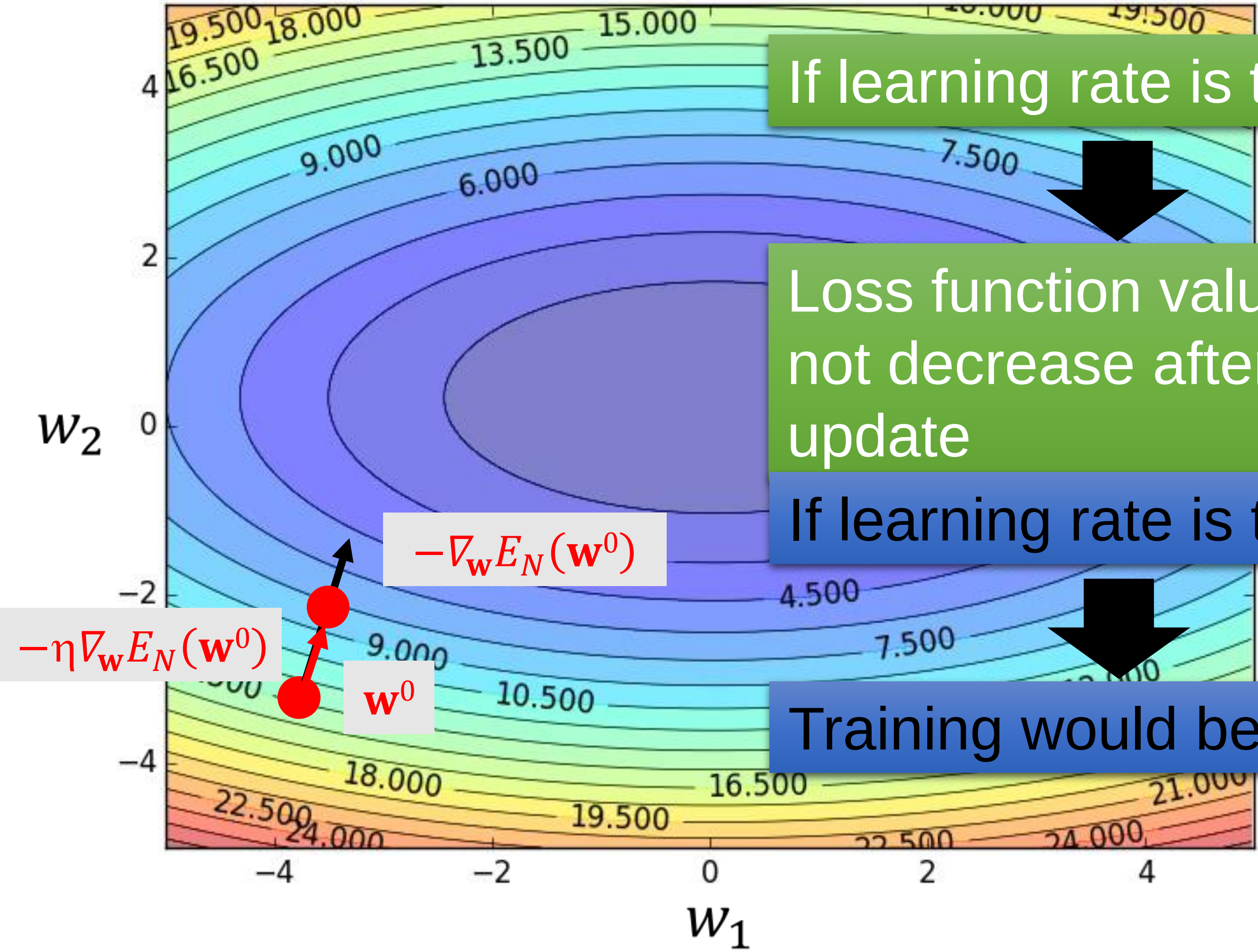
Set the learning rate η carefully

If learning rate is too large

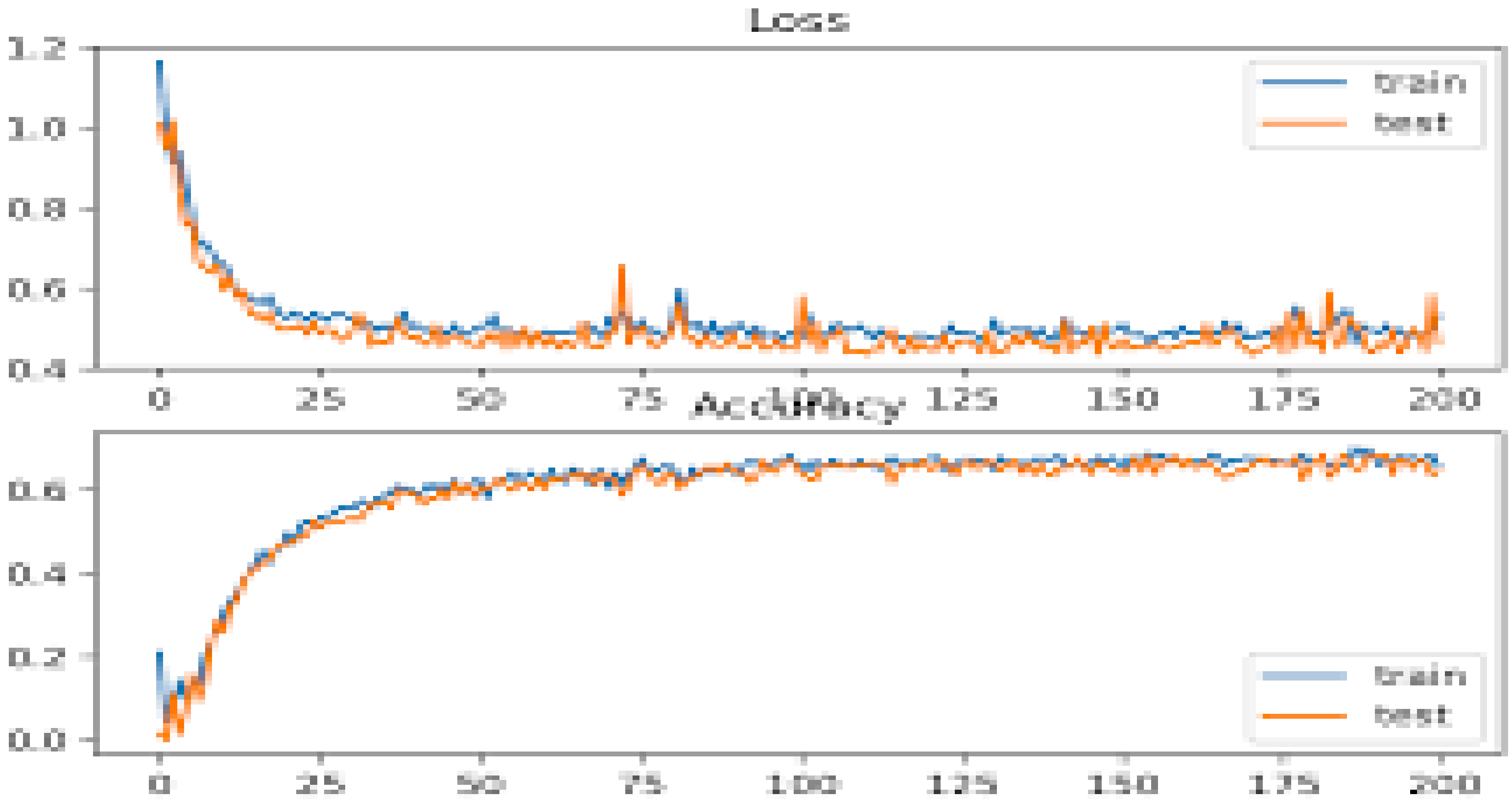
Loss function value may not decrease after each update

If learning rate is too small

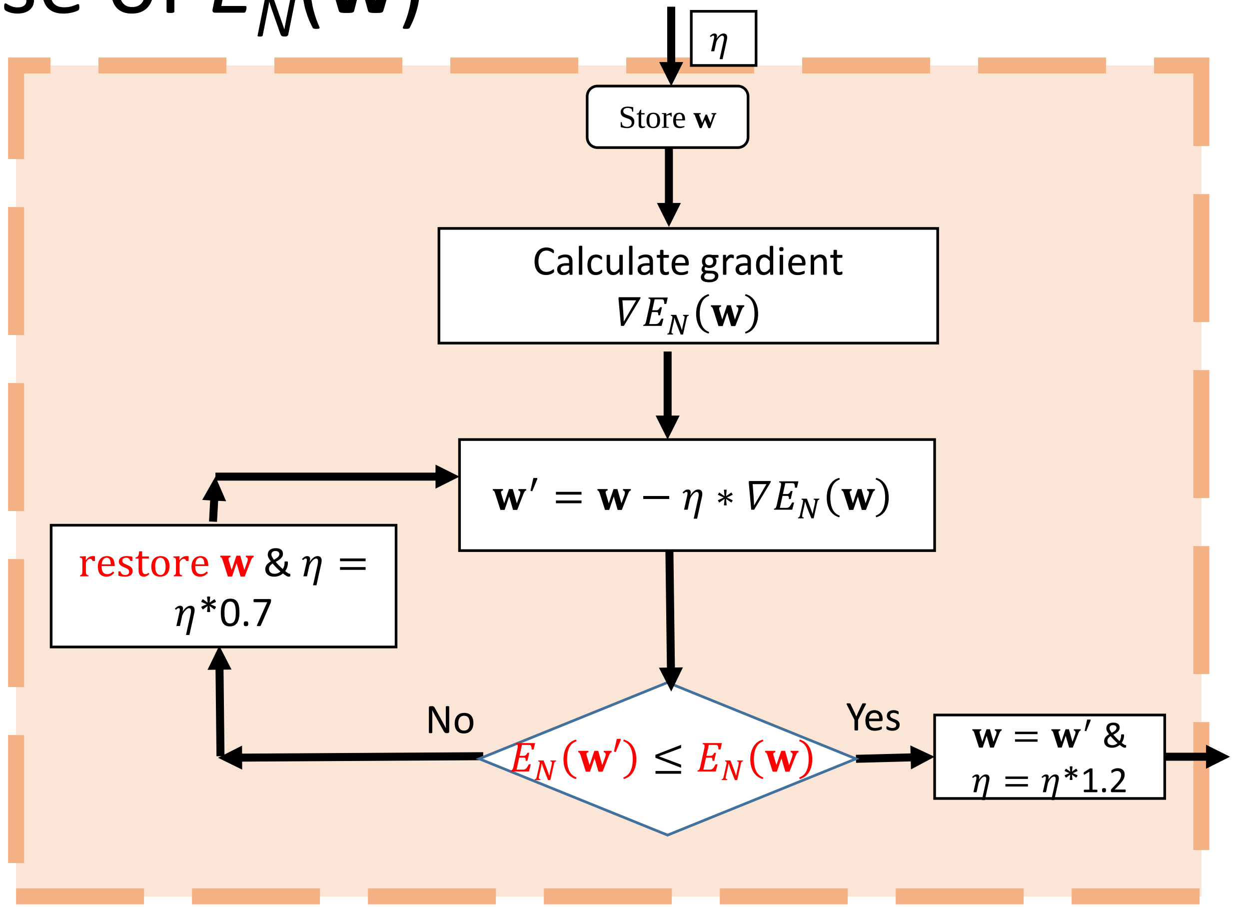
Training would be too slow



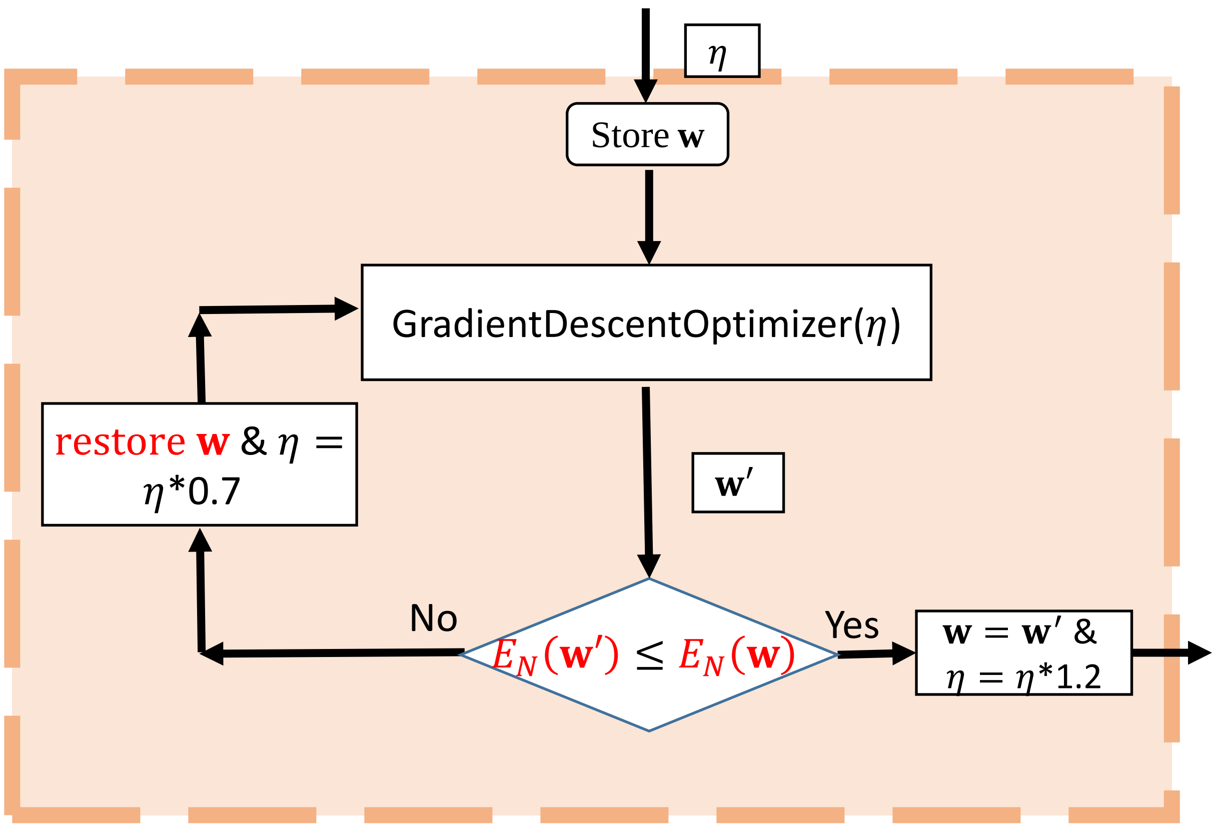
Fixed learning rate



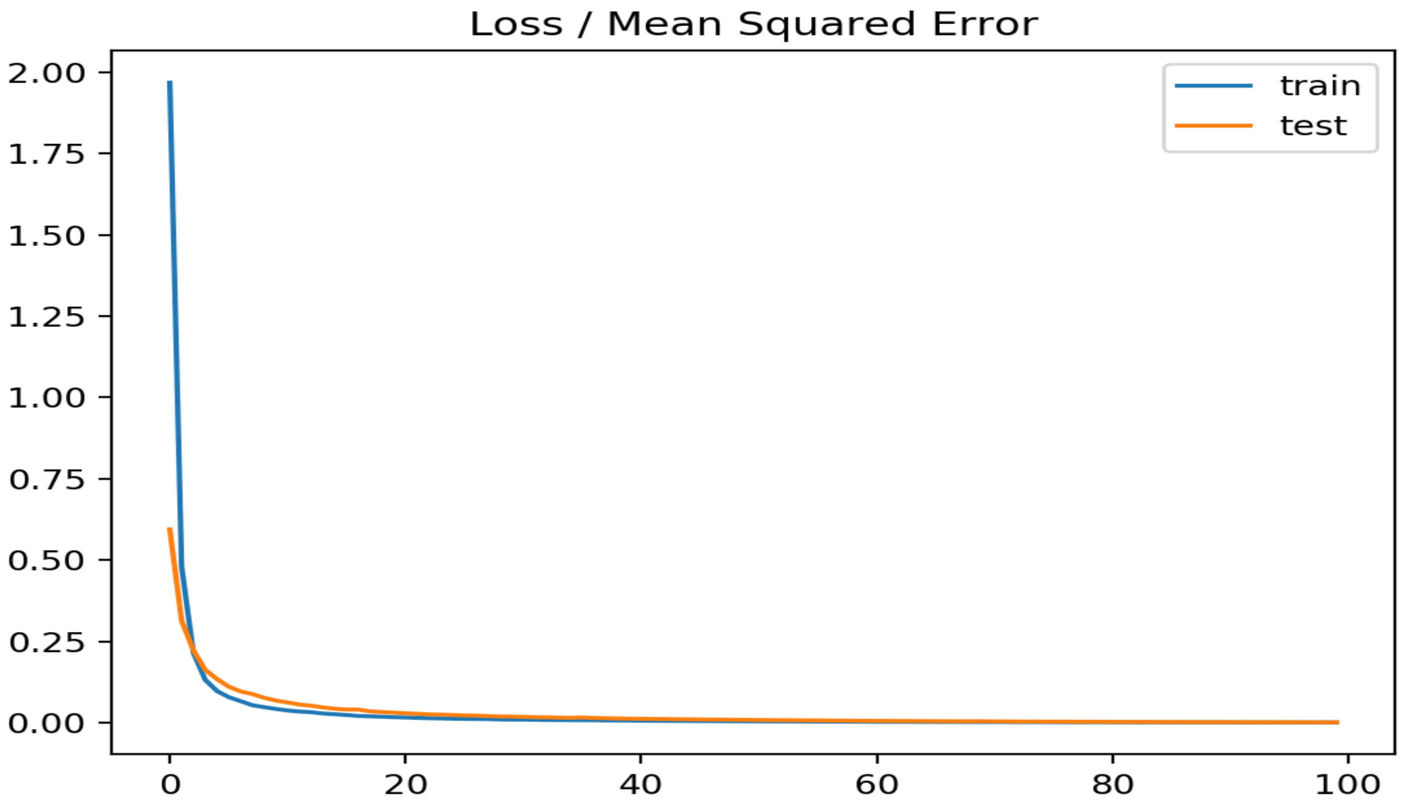
The adaptable η arrangement in the backward operation module for guaranteeing the decrease of $E_N(\mathbf{w})$



The adaptable η arrangement in the backward operation module with GradientDescentOptimizer



Adaptable learning rate



Homework #3-1

Rewrite the code of learning algorithm for 2-layer nets with the arrangement of adaptable learning rate η in the backward operation module

learning goals (and stopping criteria) 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. $|f(\mathbf{x}^c, \mathbf{w}) - y^c| < \varepsilon \ \forall\ c \ where\ \varepsilon\ is\ tiny$

Memorizing Goals and Learning Goals

Rule-based

- $\mathbf{x} \rightarrow y$
- Modeling stage ← store all pairs of ($\mathbf{x}$, y); memorizing
- Inferencing stage ← put in any $\mathbf{x}$ to get its inferencing result.

Learning-based

- $y = f(\mathbf{x})$
- Training stage ← tune weights according to pairs of ($\mathbf{x}$, y); learning
- Inferencing stage ← put in any $\mathbf{x}$ to get its inferencing result.

Memorizing Goals and Learning Goals

Rule-based

- $\mathbf{x} \rightarrow y$
- Modeling stage
- ✓ $\mathbf{x}^1 \rightarrow y^1$
- ✓ $\mathbf{x}^2 \rightarrow y^2$
- ✓ $\mathbf{x}^2 \rightarrow y^3$ and $y^3 \neq y^2$
- If $y^3 \neq y^2$, then the inferencing result of $\mathbf{x}^2$ is wrong

Learning-based

- $y = f(\mathbf{x})$
- Training stage
- ✓ $\mathbf{x}^1 \rightarrow y^1$
- ✓ $\mathbf{x}^2 \rightarrow y^2$
- ✓ $\mathbf{x}^2 \rightarrow y^3$ and $y^3 \neq y^2$
- The inferencing result $f(\mathbf{x}^2)$? Depends on the learning goal.

Memorizing Goals and Learning Goals

Rule-based

- $\mathbf{x} \rightarrow y$
- Modeling stage

✓ $\mathbf{x}^1 \rightarrow y^1$

✓ $\mathbf{x}^2 \rightarrow y^2$

✓ $\mathbf{x}^2 \rightarrow y^3$ and $y^3 \neq y^2$

- If $y^3 \neq y^2$, then the inferencing result of $\mathbf{x}^2$ is wrong

Learning-based

- $y = f(\mathbf{x})$
- Training stage

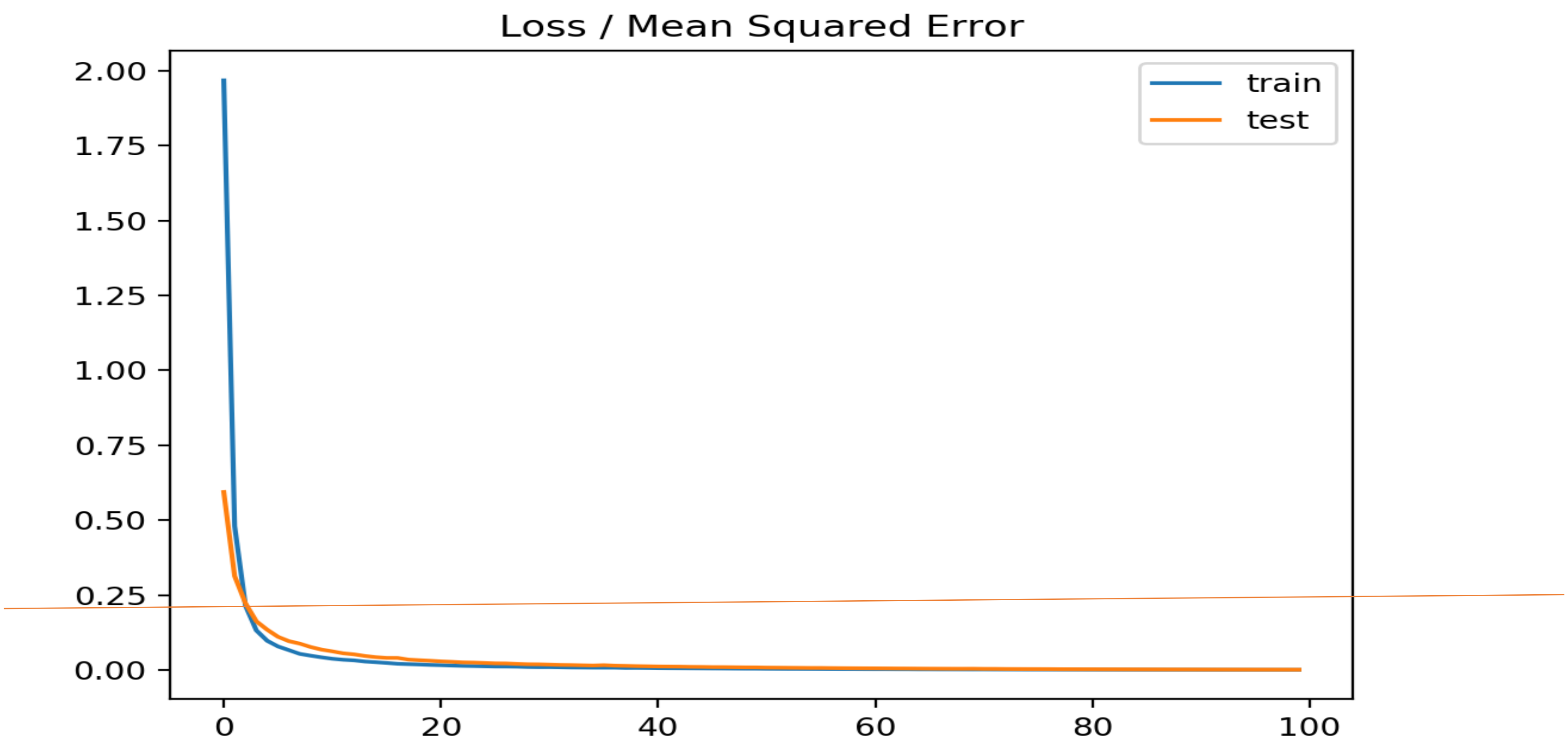
✓ $\mathbf{x}^1 \rightarrow y^1$

✓ $\mathbf{x}^2 \rightarrow y^2$

✓ $\mathbf{x}^2 \rightarrow y^3$ and $y^3 \neq y^2$

- If the learning goal is $E_N(\mathbf{w}) = 0$, then the *training cannot be accomplished.*
- If the learning goal is $E_N(\mathbf{w}) > 0$ that allows $f(\mathbf{x}^2) = (y^3 + y^2)/2$, then $f(\mathbf{x}^2) = (y^3 + y^2)/2$
- Or …

The learning goal

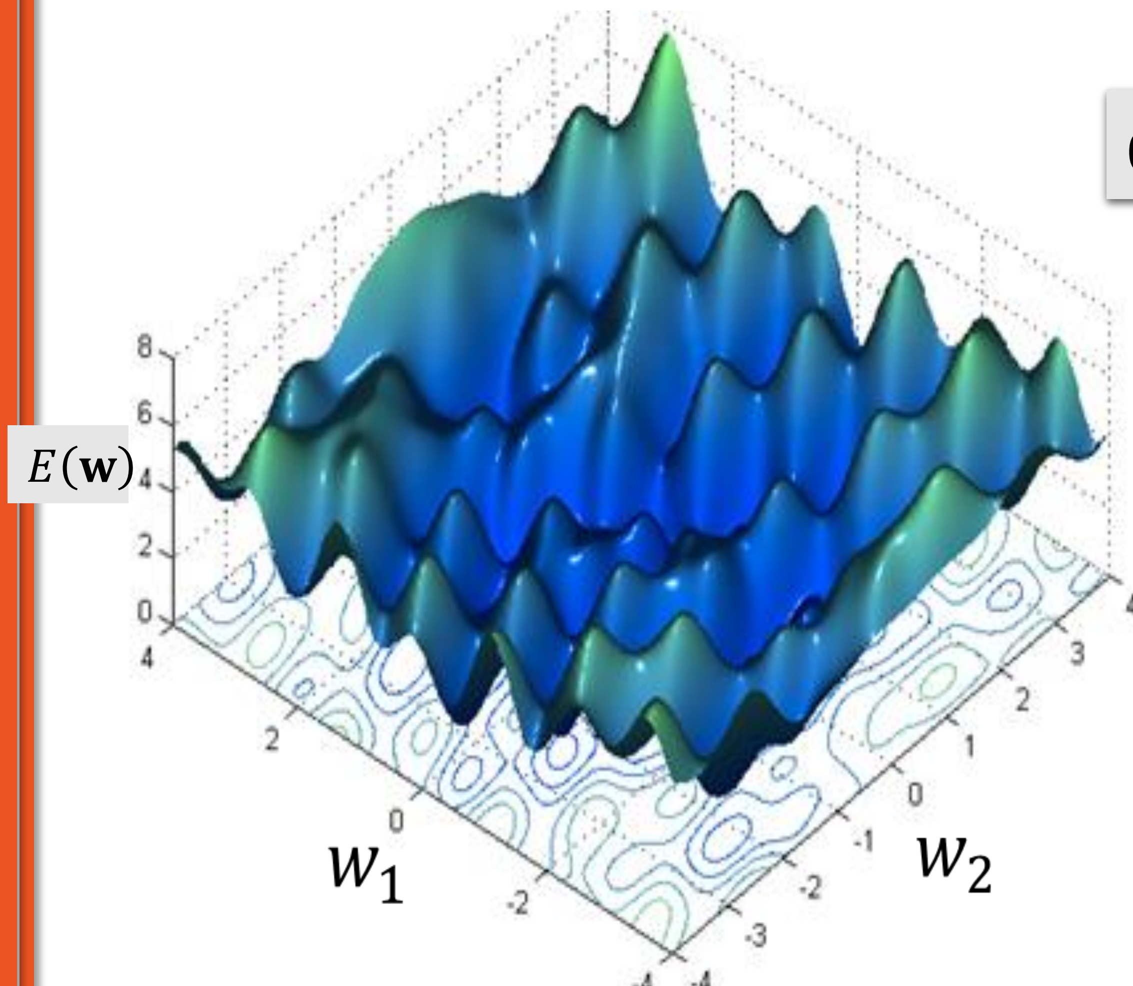


Stopping criteria (and also learning goals) for the learning

The learning process should stop when

1. Hit the epoch constraint (e.g., i >= 100)
2. ~~$E_{\mathbb{N}}(\mathbf{w}) = 0$~~
3. ~~Obtain a tiny $E_{\mathbb{N}}(\mathbf{w})$ value~~
4. $|f(\mathbf{x}^c, \mathbf{w}) - y^c| < \varepsilon \ \forall\ c$ *where* ε *is tiny*
5. … ???

Optimizers never guarantee global minimum

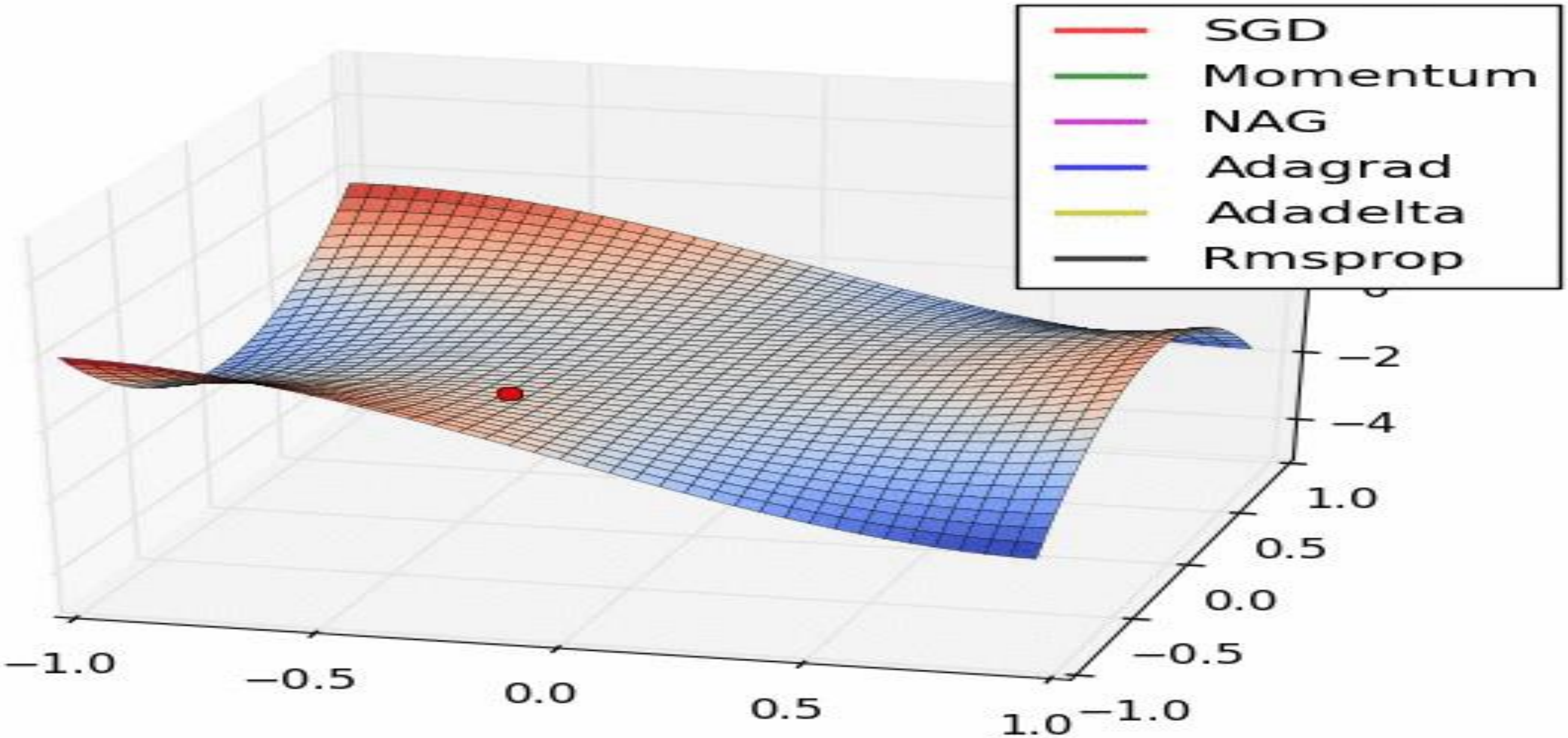


different initial weights

⬇

different minimum

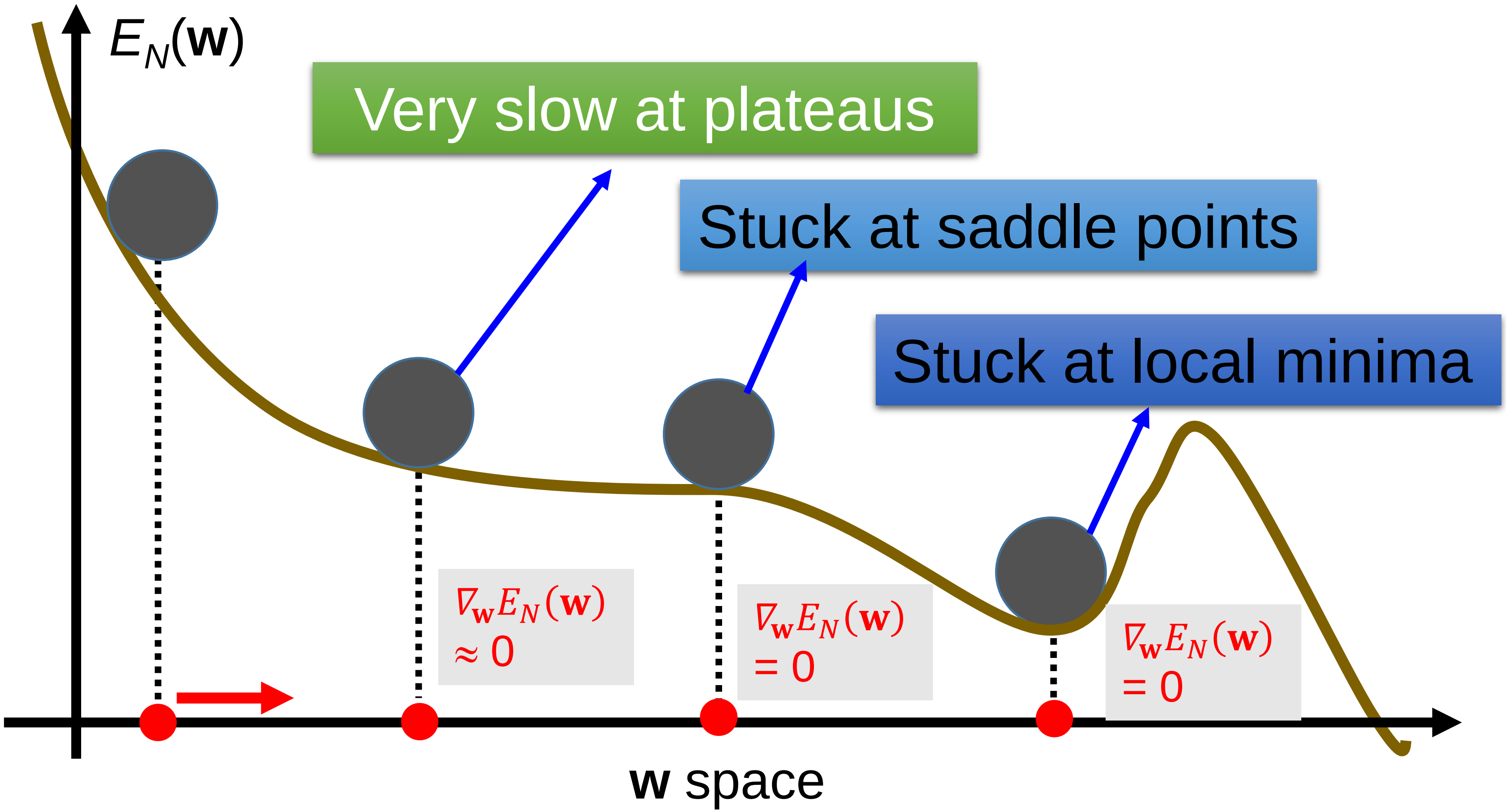
The learning process with different optimizers



Reference:

1. https://en.wikipedia.org/wiki/Test_functions_for_optimization : Beale function
2. An overview of gradient descent optimization algorithms.pdf

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



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

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

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

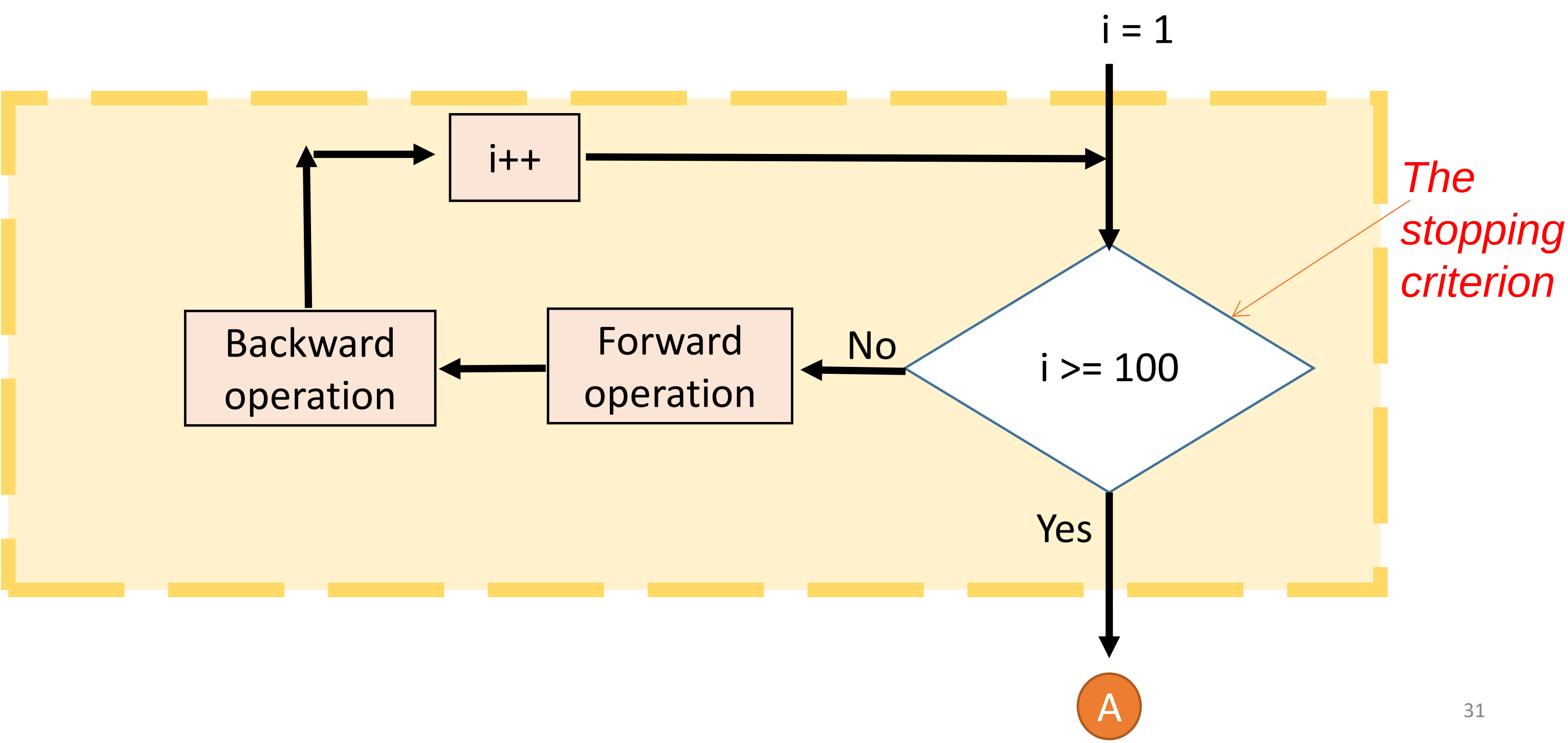
1. ~~The learning process should stop when~~ $\cancel{\|\nabla_{\mathbf{w}} E_N(\mathbf{w})\| = 0.}$
2. The learning process should stop when $\|\nabla_{\mathbf{w}} E_N(\mathbf{w})\|$ is tiny.
3. The learning process should stop when η (the adaptive learning rate) is tiny.

The undesired attractors:

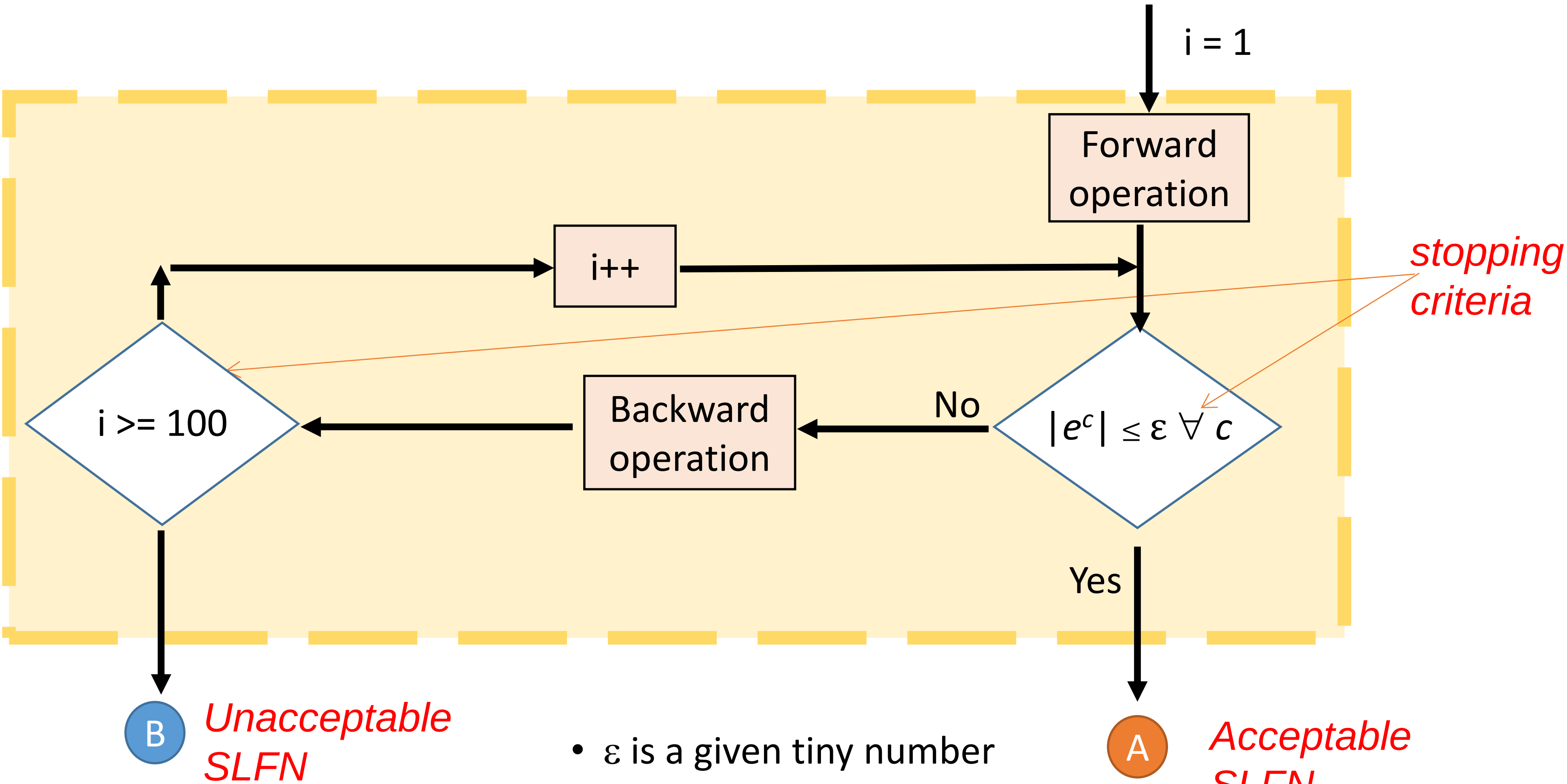
a) the local optimum/the saddle point/the plateau

b) the global optimum of the defective network architecture

The algorithm without extra stopping criteria

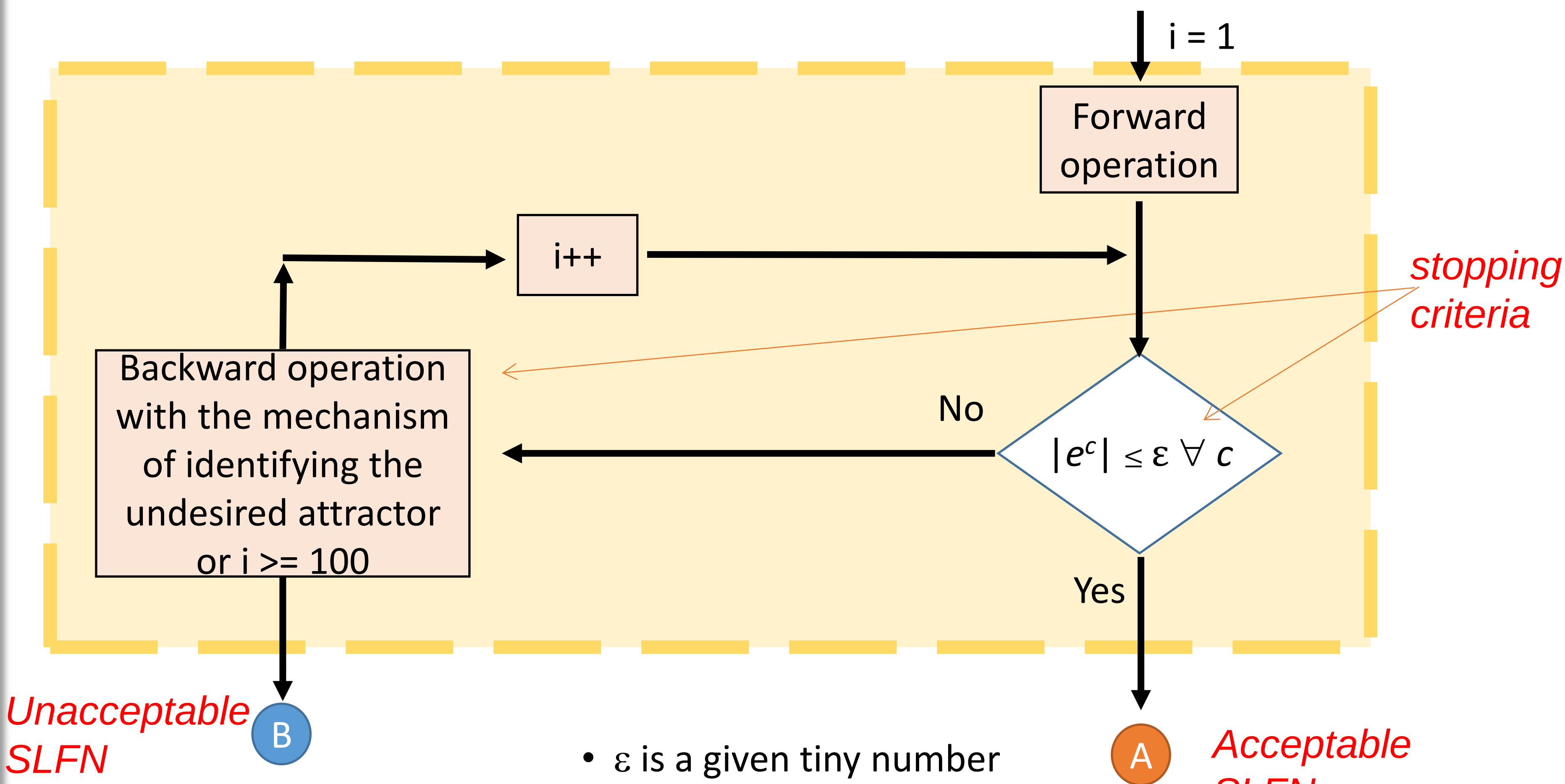


The algorithm with an extra stopping criterion that indicates either an undesired SLFN or a desired SLFN

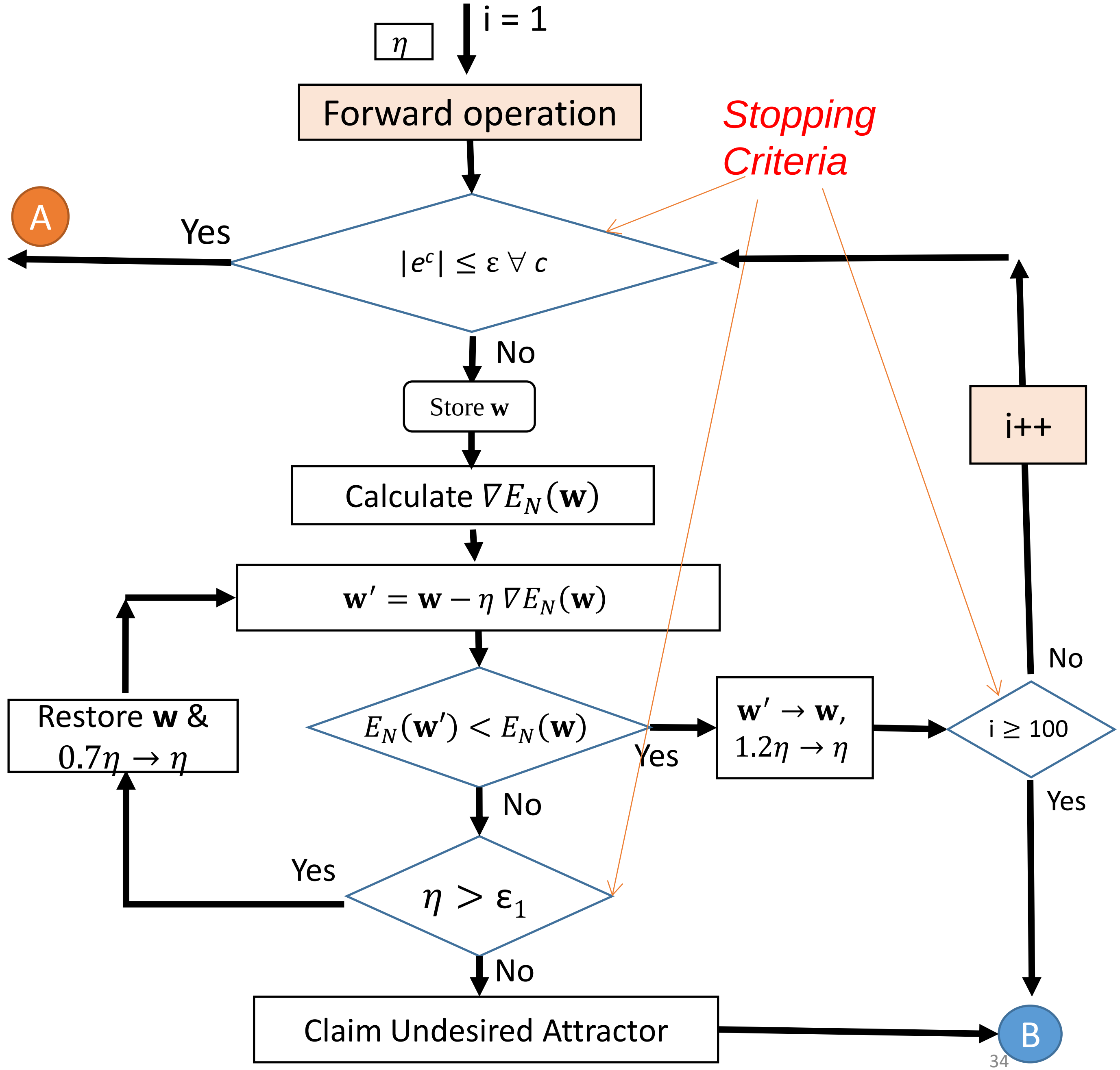


- ε is a given tiny number
- $e^c = f(\mathbf{x}^c, \mathbf{w}) - y^c$

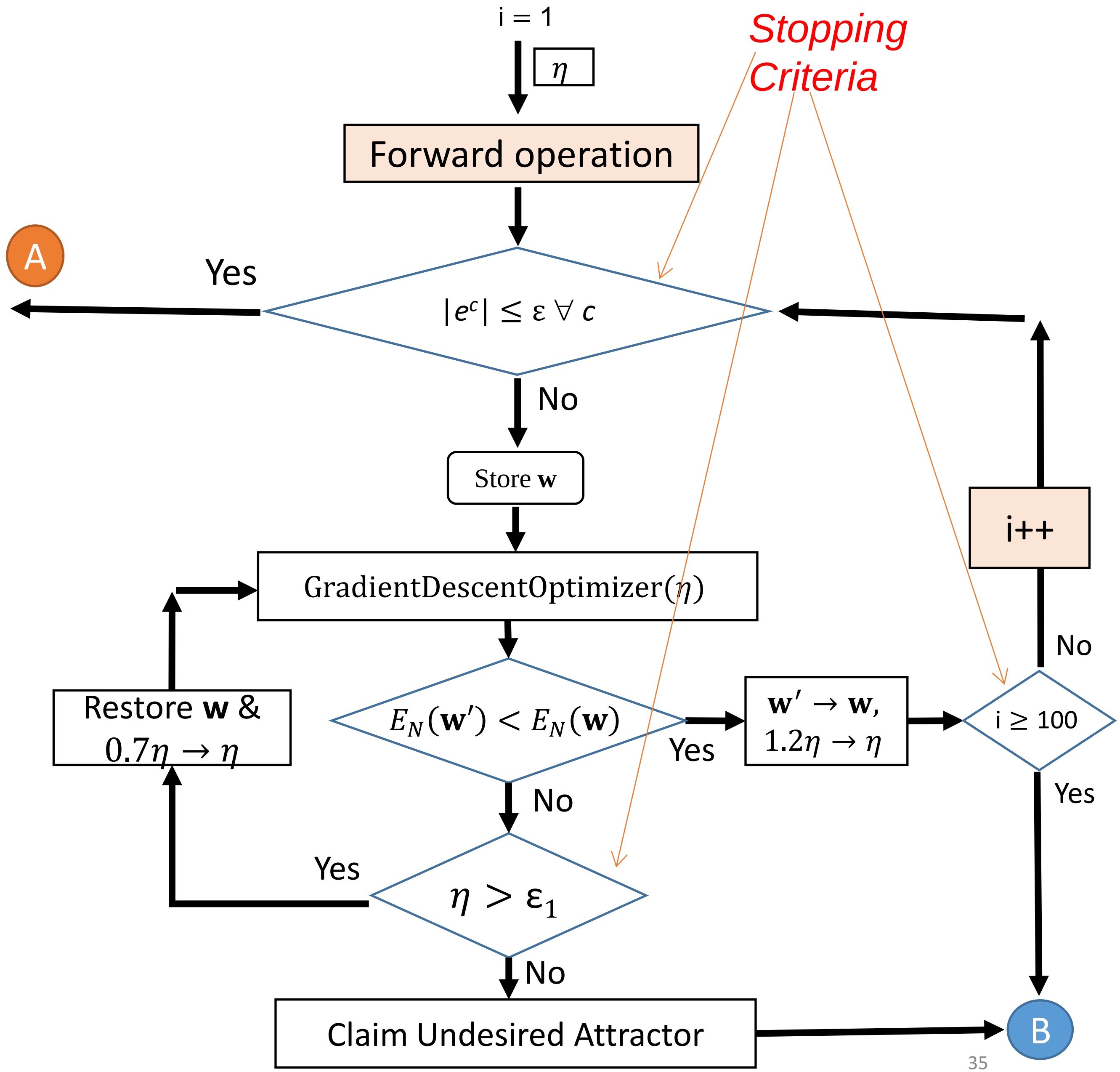
The algorithm with extra stopping criteria



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

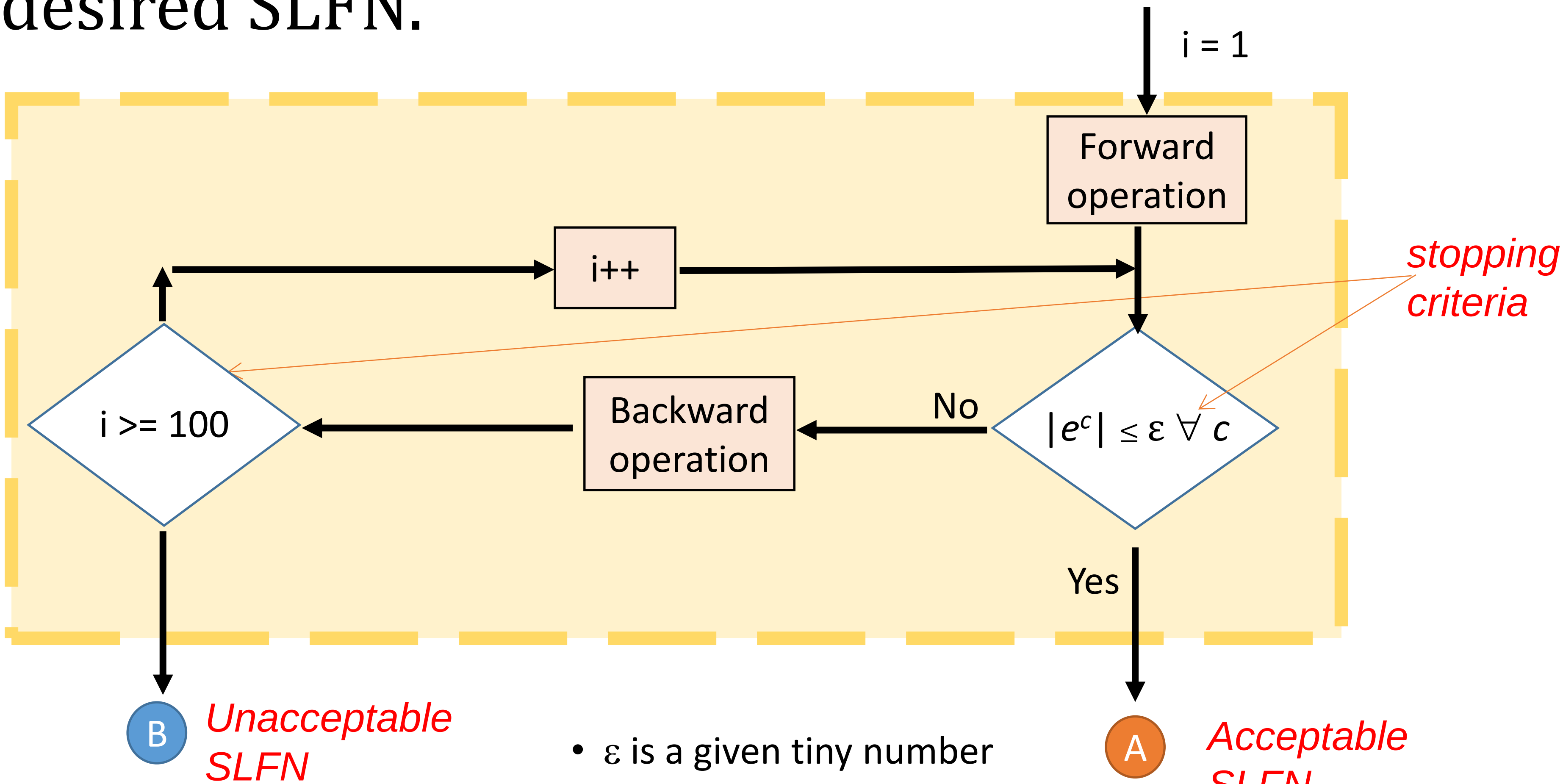


- ε and ε_1 are given tiny numbers
- $e^c = f(\mathbf{x}^c, \mathbf{w}) - y^c$



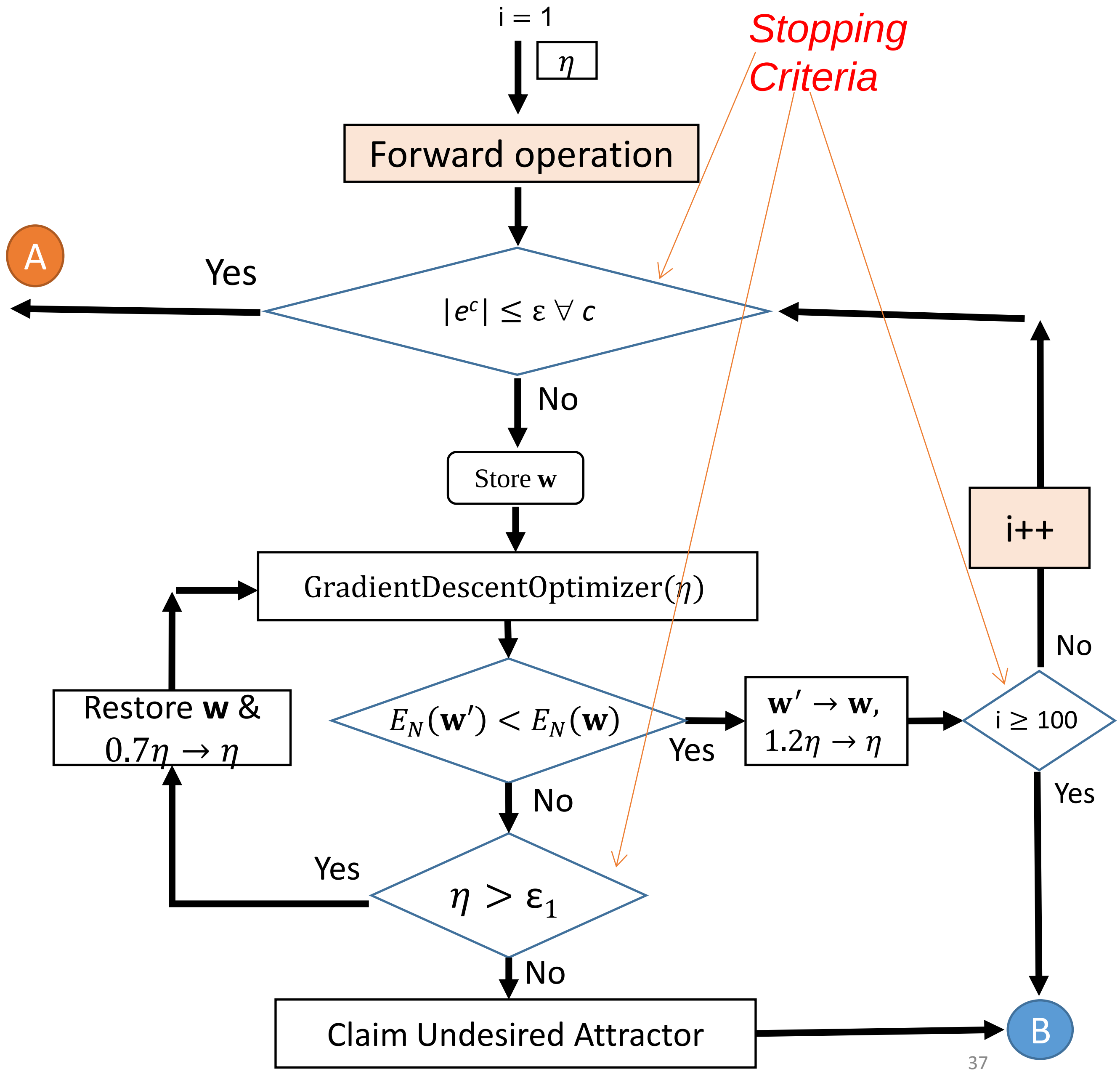
- ε and ε_1 are given tiny numbers
- $e^c = f(\mathbf{x}^c, \mathbf{w}) - y^c$

Homework #3-2a: Rewrite the code of learning algorithm for SLFNs that indicates whether the final result is either an undesired SLFN or a desired SLFN.



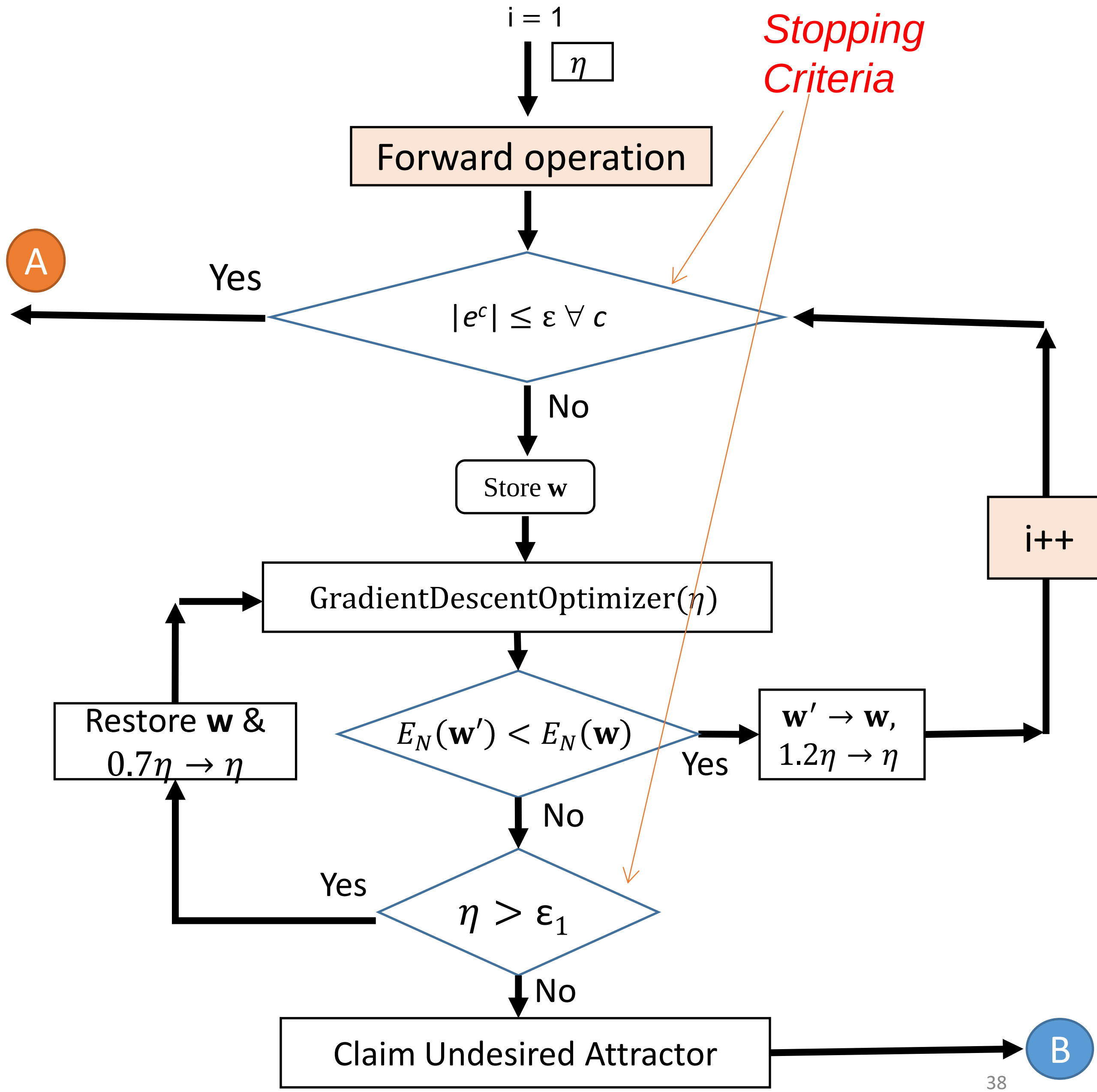
- ε is a given tiny number
- $e^c = f(\mathbf{x}^c, \mathbf{w}) - y^c$

Homework #3-2b: Rewrite the code of learning 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
- $e^c = f(\mathbf{x}^c, \mathbf{w}) - y^c$

Homework #3-2c: Rewrite the code of learning 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
- $e^c = f(\mathbf{x}^c, \mathbf{w}) - y^c$