

# Deep Learning in Bioinformatics

*Practice*

DATA MINING & BIOINFORMATICS LAB.

# Index

---

- 1. Deep Learning Process**
- 2. Sequence Classification**
- 3. Link Prediction (Graph Neural Network)**

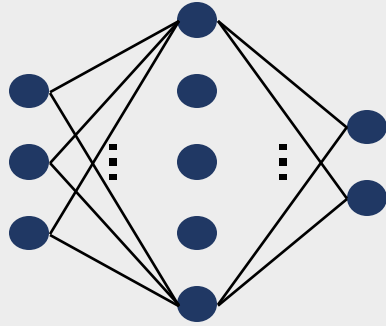
# Deep Learning Process

## Step1



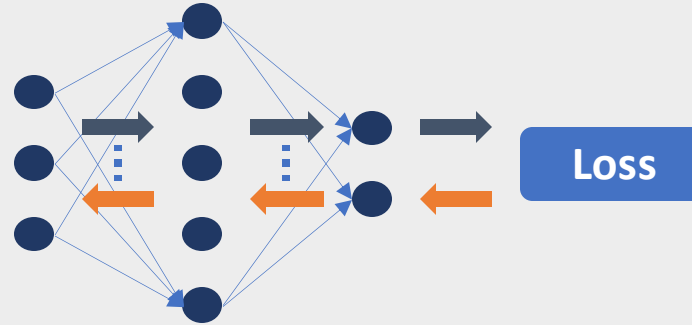
- Collecting data
- Preprocessing
- Dataset
- Data-loader

## Step2



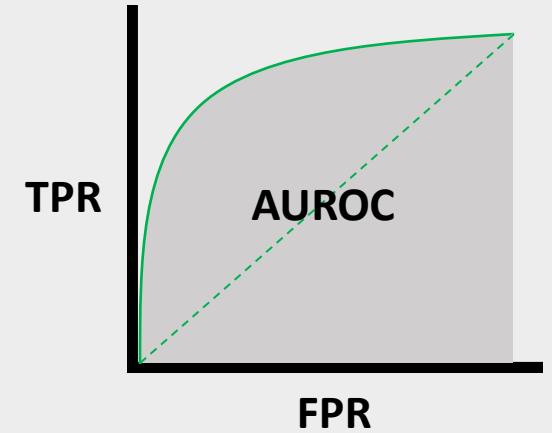
- Selecting method
- Building model

## Step3



- Hyper-parameters
- Loss function
- Optimizer
- Training

## Step4



- Predicting
- Evaluating

# Sequence Classification

## Dataset

**DeepLoc 2.0** provides proteins categorized into one of multiple of these ten locations

: Cytoplasm, Nucleus, Extracellular, Cell membrane, Mitochondrion, Plastid, Endoplasmic reticulum, Lysosome/Vacuole, Golgi apparatus, Peroxisome.

| ACC    | Kingdom | Partition | Membrane | Cytoplasm | Nucleus | ... | Sequence            |
|--------|---------|-----------|----------|-----------|---------|-----|---------------------|
| Q28165 | Metazoa | 4         | 0        | 1         | 1       | ... | MAAAAAAAAAAAGAAG... |
| Q86U42 | Metazoa | 4         | 0        | 1         | 1       | ... | MAAAAAAAAAAAGAAG... |
| Q0GA42 | Metazoa | 3         | 1        | 0         | 0       | ... | MAAAAAAAAAALGVRL... |
| P82349 | Metazoa | 1         | 1        | 1         | 0       | ... | MAAAAAAAAAATEQQG... |
| Q7L5N1 | Metazoa | 1         | 0        | 1         | 1       | ... | MAAAAAAAAAATNGTG... |
| Q96S94 | Metazoa | 0         | 0        | 0         | 1       | ... | MAAAAAAAGAAGSAA...  |
| Q9CQ25 | Metazoa | 0         | 0        | 1         | 0       | ... | MAAAAAAAGGAALAV...  |
| Q96P70 | Metazoa | 4         | 0        | 1         | 1       | ... | MAAAAAAGAASGLPG...  |
| P63086 | Metazoa | 1         | 1        | 1         | 0       | ... | MAAAAAAGPEMVRGQ...  |

# Dataset

---

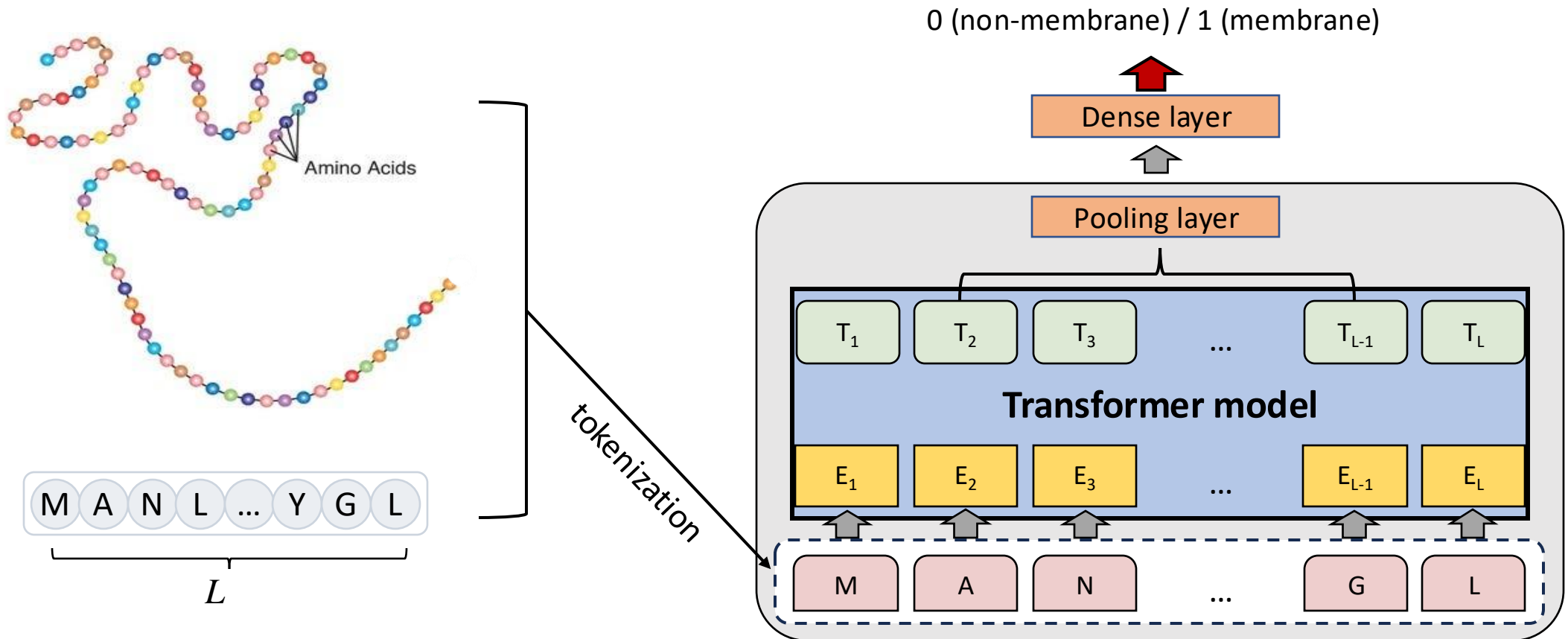
## Toy dataset – DeepLoc2.0\_membrane

| index | acc    | sequence           | label |
|-------|--------|--------------------|-------|
| 0     | P63086 | MAAAAAAGPEMVRGQ... | 1     |
| 1     | Q9UID3 | MAAAAAAGPSPGSGP... | 0     |
| 2     | P51788 | MAAAAAEEGMEPRAL... | 1     |
| 3     | Q3UCQ1 | MAAAAALSGAGAPPA... | 0     |
| 4     | Q9NVF7 | MAAAAEERMAEEGGG... | 0     |
| 5     | Q8CD10 | MAAAAGRSAWLAAWG... | 1     |
| 6     | Q5VY80 | MAAAAIPALLLCLPL... | 1     |
| 7     | Q96IV0 | MAAAALGSSSGSASP... | 0     |
| 8     | O35094 | MAAAALRGGWCRCPR... | 1     |
| 9     | Q9Y584 | MAAAAPNAGGSAPET... | 1     |
| 10    | Q9NXG6 | MAAAAVTGQRPETAA... | 1     |
| ...   | ...    | ...                | ...   |

# Model

- **Protein sequence binary classification**

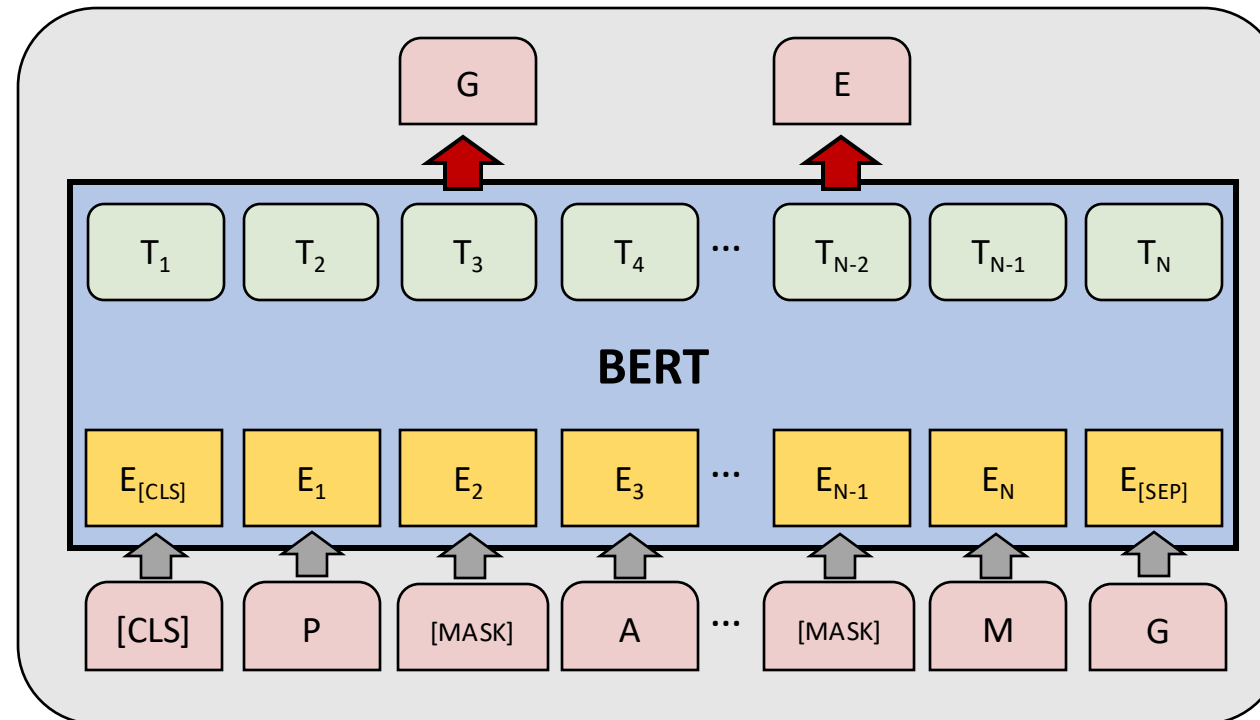
- Sequence classification is the task of assigning a label or class to a given text.
- Input : protein sequence
- Target : binary classification (membrane or non-membrane)



# Model

## Pre-training model – ProtBERT

- *"Prottrans: Toward understanding the language of life through self-supervised learning."*
- Bert-based protein language model developed as part of the *Prottrans* project.
- ProtBERT is trained in self-supervised manner, essentially learning to predict masked amino acids (tokens) in already known sequences.

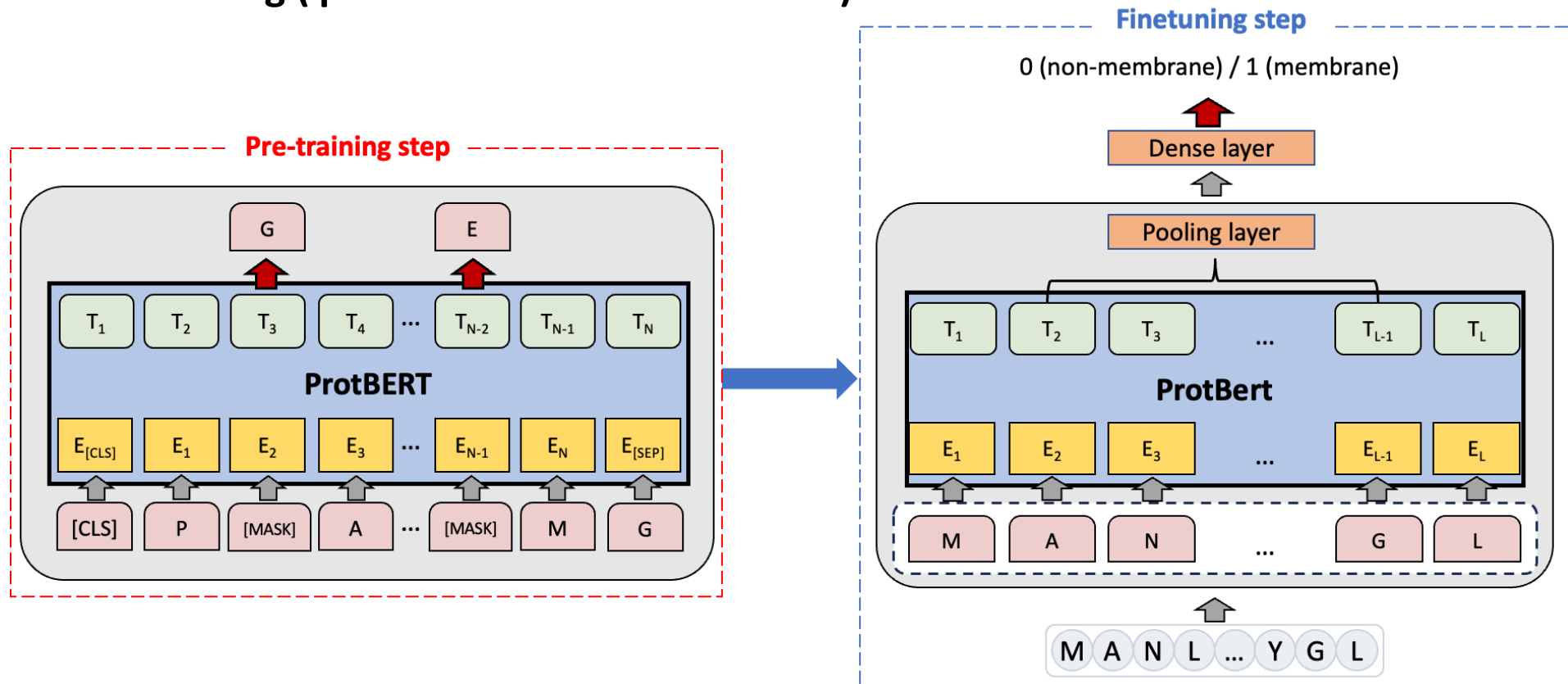




# Model

- **Protein sequence binary classification**

- Sequence classification is the task of assigning a label or class to a given text.
- Input : protein sequence
- Target : binary classification (membrane or non-membrane)
- **Transfer learning ( pretrained model : ProtBERT)**



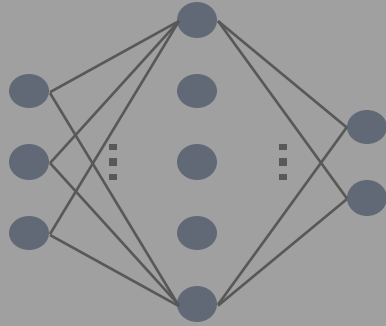
# Deep Learning Process

## Step1



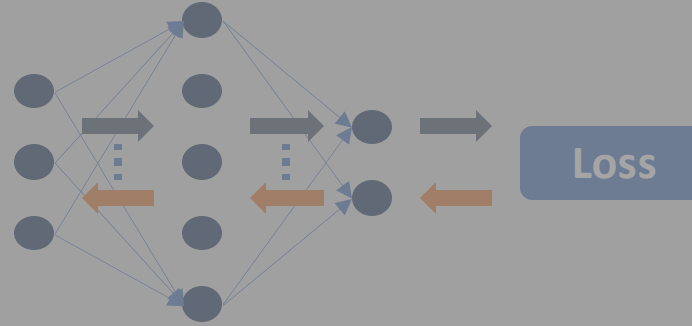
- Collecting data
- Preprocessing
- Dataset
- Data-loader

## Step2



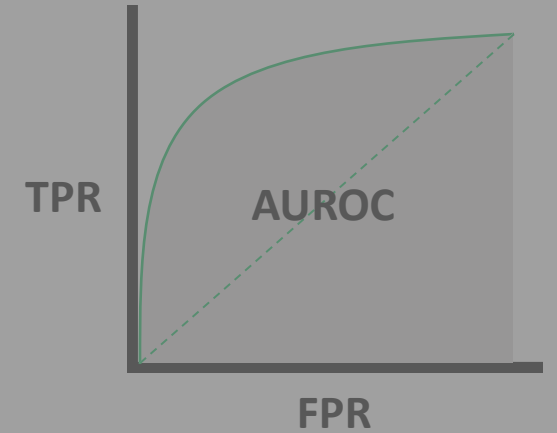
- Selecting method
- Building model

## Step3



- Hyper-parameters
- Loss function
- Optimizer
- Training

## Step4



- Predicting
- Evaluating

## Step 1. Load dataset

---

```
1 ### Load dataset ###
2 train_df = pd.read_csv('./toy_train.csv')
3 valid_df = pd.read_csv('./toy_valid.csv')
4
5 train_df.head()
```

|   | index | acc    | sequence                                          | labels |
|---|-------|--------|---------------------------------------------------|--------|
| 0 | 0     | P63086 | MAAAAAAGPEMVRGQVFDVGPRTNLSYIGEGAYGMVCSAYDNLNK...  | 1      |
| 1 | 1     | Q9UID3 | MAAAAAAGPSPGSGPGDSPEGPEGEAPERRRKAHGMLKLYYGLSEG... | 0      |
| 2 | 2     | P51788 | MAAAAAEEGMEPRALQYEQTLMYGRYTQDLGAFAKEEAARIRLGGP... | 1      |
| 3 | 3     | Q3UCQ1 | MAAAAALSGAGAPPAGGGAGGGGSPPGGWAVARLEGREFEYLMKKR... | 0      |
| 4 | 4     | Q9NVF7 | MAAAAEERMAEEGGGGQGDGGSSLASGSTQRQPPPPAPQHPQPGSQ... | 0      |

## Step 1. Load dataset

- Dataset은 데이터를 효율적으로 관리하고, 정형화된 방식으로 데이터를 가져오기 위함
- `__getitem__`과 `__len__`을 구현하여 데이터 접근과 크기 확인이 용이함
- 다양한 전처리를 쉽게 적용할 수 있음

```
# Dataset 클래스 정의
class ProteinDataset(Dataset):
    def __init__(self, data):
        self.sequences = [" ".join(list(re.sub(r"[UZOB]", "X", sequence))) for sequence in data['sequence'].tolist()]
        self.labels = data['labels']

    def __len__(self):
        return len(self.sequences)

    def __getitem__(self, idx):
        return {
            "sequence" : self.sequences[idx],
            "label": self.labels[idx]
        }
```

```
1 train_dataset = ProteinDataset(train_df)
2 valid_dataset = ProteinDataset(valid_df)
```

## Step 1. Load dataset

---

- DataLoader는 Batch 단위로 데이터를 나누어 효율적인 배치 학습을 가능하게 함

```
1 train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
2 valid_loader = DataLoader(valid_dataset, batch_size=batch_size, shuffle=False)
```



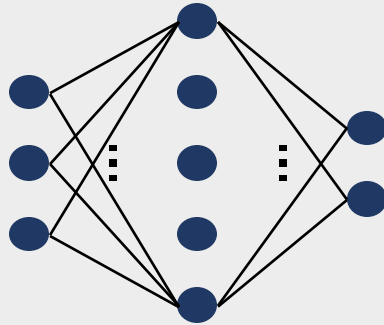
# Deep Learning Process

## Step1



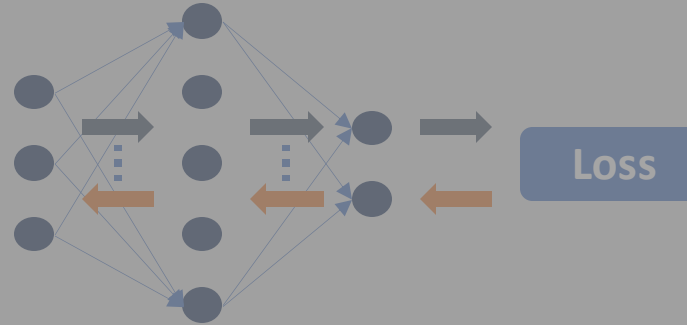
- Collecting data
- Preprocessing
- Dataset
- Data-loader

## Step2



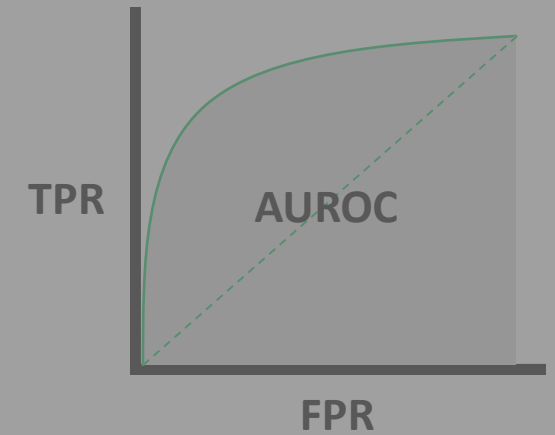
- Selecting method
- Building model

## Step3



- Hyper-parameters
- Loss function
- Optimizer
- Training

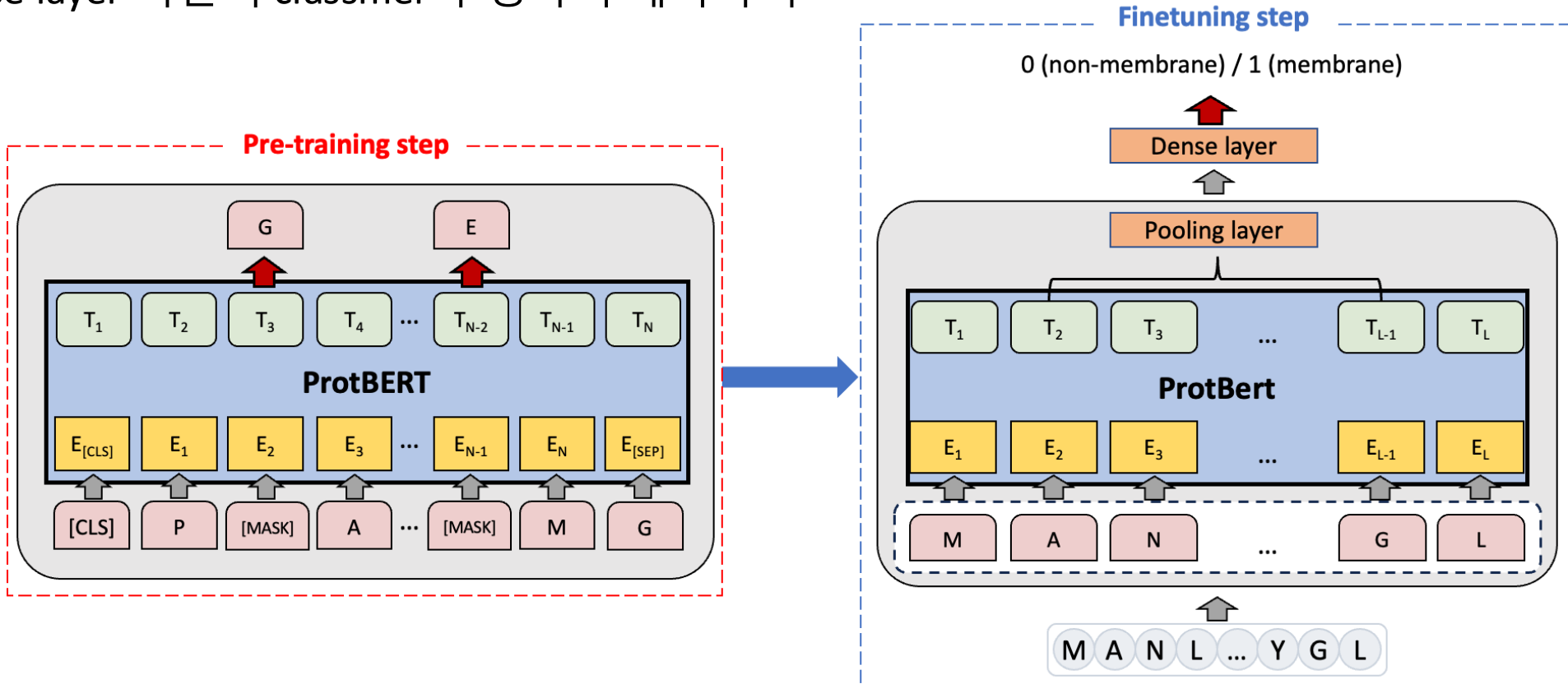
## Step4



- Predicting
- Evaluating

## Step 2. Build model for finetuning

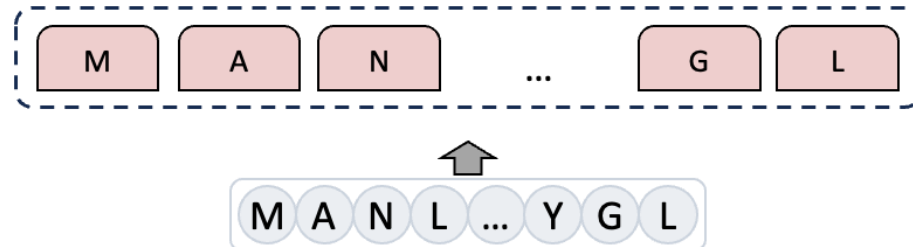
- Protein sequence를 tokenization 하기
- Pretrained ProtBERT 불러오기
- 불러온 protBERT를 사용하여 amino-acid embedding 만들기
- Protein-level embedding을 위하여 amino-acid embedding들을 pooling하기
- Dense layer 기반의 classifier 구성하여 예측하기



## Step 2. Build model for finetuning

- Protein sequence를 tokenization 하기
- Pretrained ProtBERT 불러오기
- 불러온 ProtBERT를 사용하여 amino-acid embedding 만들기
- Protein-level embedding을 위하여 amino-acid embedding들을 pooling하기
- Dense layer 기반의 classifier 구성하여 예측하기

```
5 self.tokenizer = BertTokenizer.from_pretrained(pretrained_model_name)
    :
25 inputs = self.tokenizer.batch_encode_plus(
26     sequences, add_special_tokens = False,
27     max_length = 512,
28     padding=True,
29     truncation=True
30 )
```

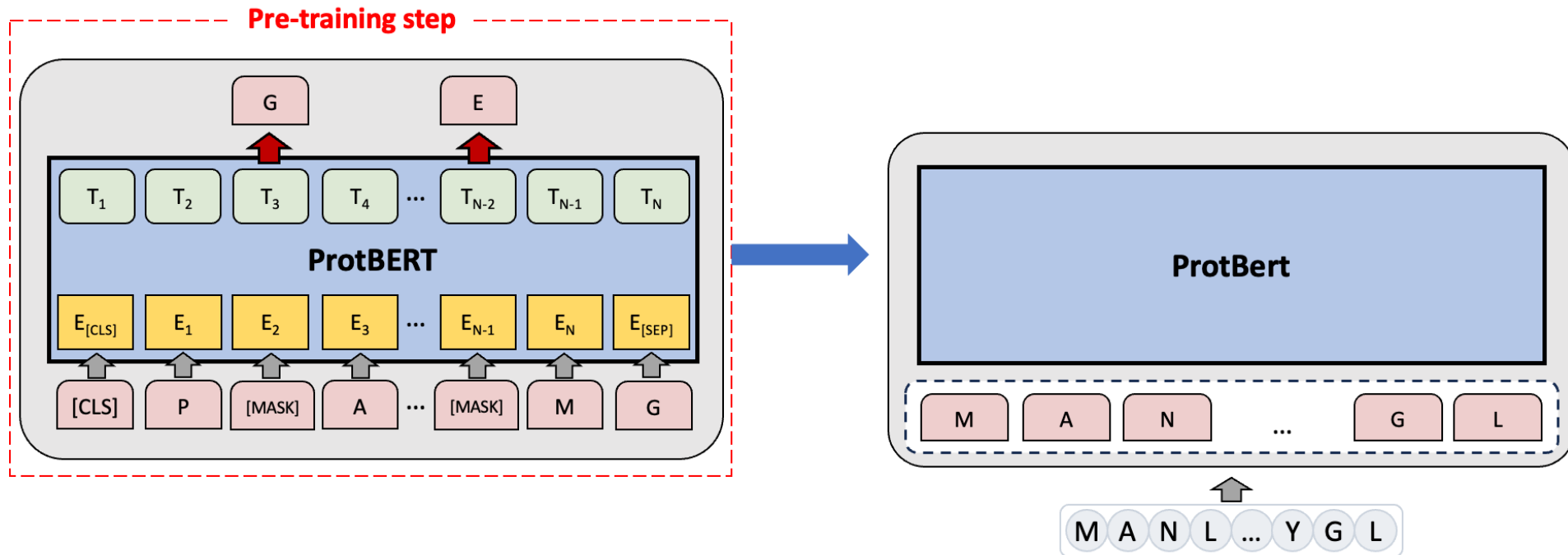




## Step 2. Build model for finetuning

- Protein sequence를 tokenization 하기
- **Pretrained ProtBERT 불러오기**
- 불러온 ProtBERT를 사용하여 amino-acid embedding 만들기
- Protein-level embedding을 위하여 amino-acid embedding들을 pooling하기
- Dense layer 기반의 classifier 구성하여 예측하기

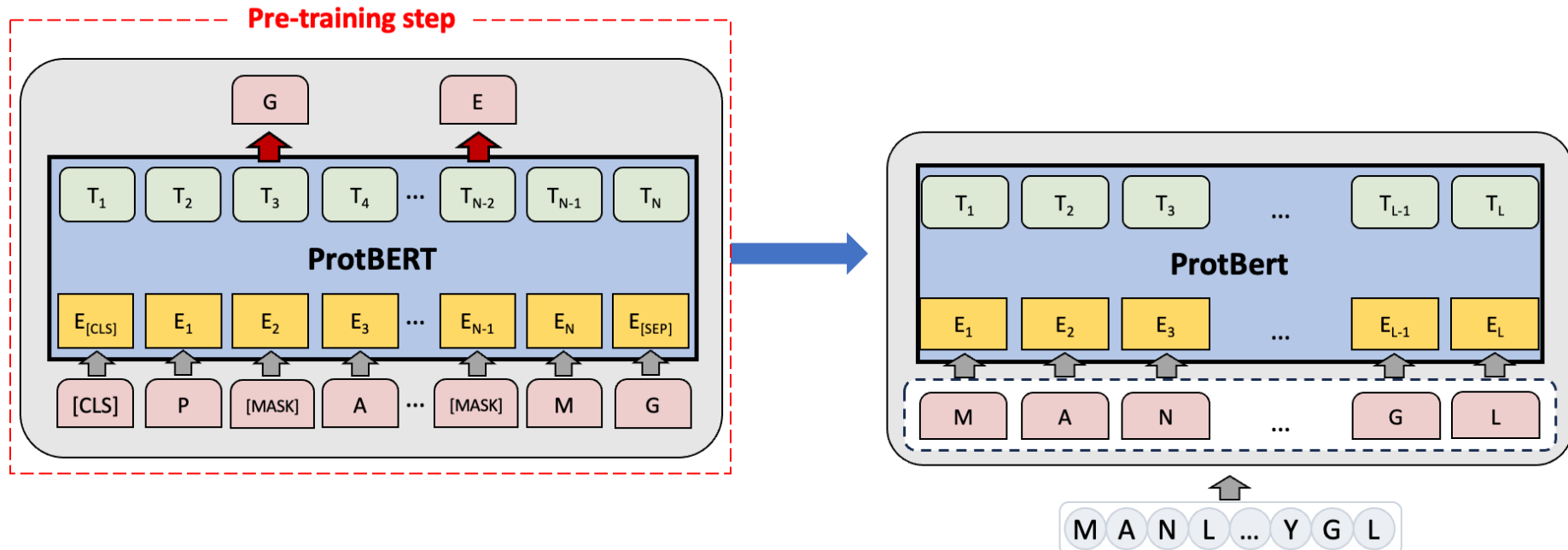
```
6 self.encoder = BertModel.from_pretrained(pretrained_model_name)
```



## Step 2. Build model for finetuning

- Protein sequence를 tokenization 하기
- Pretrained ProtBERT 불러오기
- **불러온 ProtBERT를 사용하여 amino-acid embedding 만들기**
- Protein-level embedding을 위하여 amino-acid embedding들을 pooling하기
- Dense layer 기반의 classifier 구성하여 예측하기

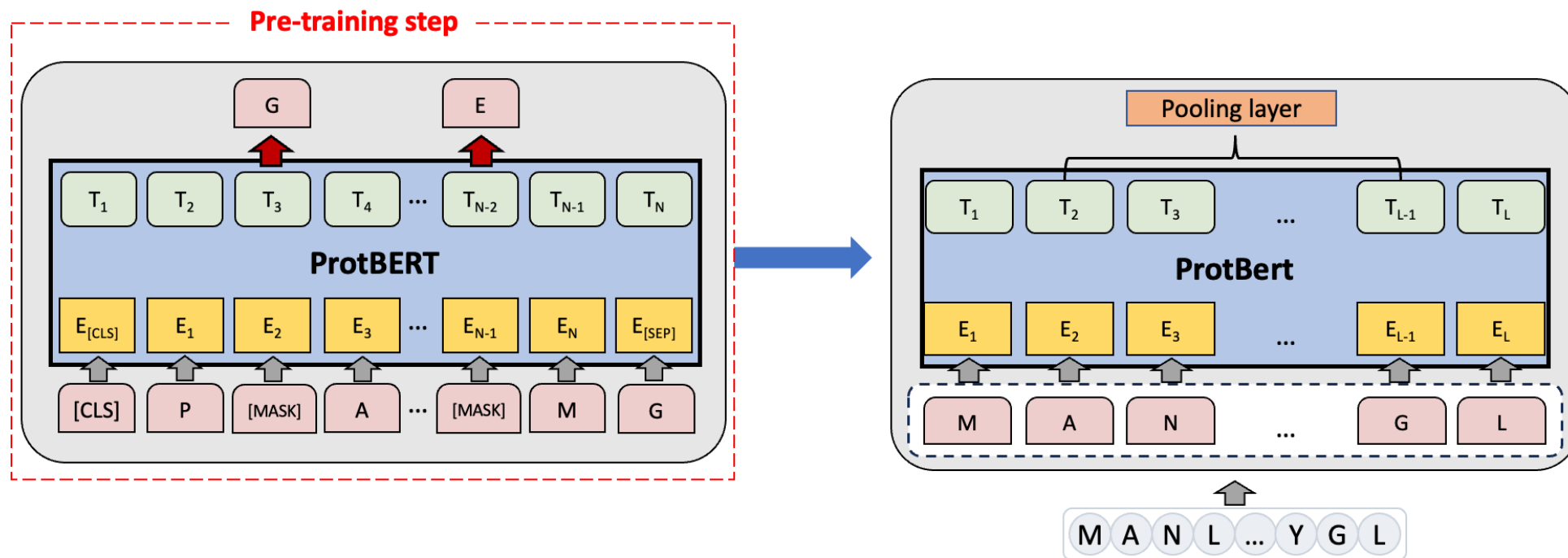
```
32 embeddings = self.encoder(  
33     input_ids=torch.tensor(inputs["input_ids"]).to(self.device),  
34     attention_mask=torch.tensor(inputs["attention_mask"]).to(self.device)  
35 )
```



## Step 2. Build model for finetuning

- Protein sequence를 tokenization 하기
- Pretrained ProtBERT 불러오기
- 불러온 ProtBERT를 사용하여 amino-acid embedding 만들기
- **Protein-level embedding을 위하여 amino-acid embedding들을 pooling하기**
- Dense layer 기반의 classifier 구성하여 예측하기

```
36 pooled_embeddings = torch.mean(embeddings[0], dim=1)
```

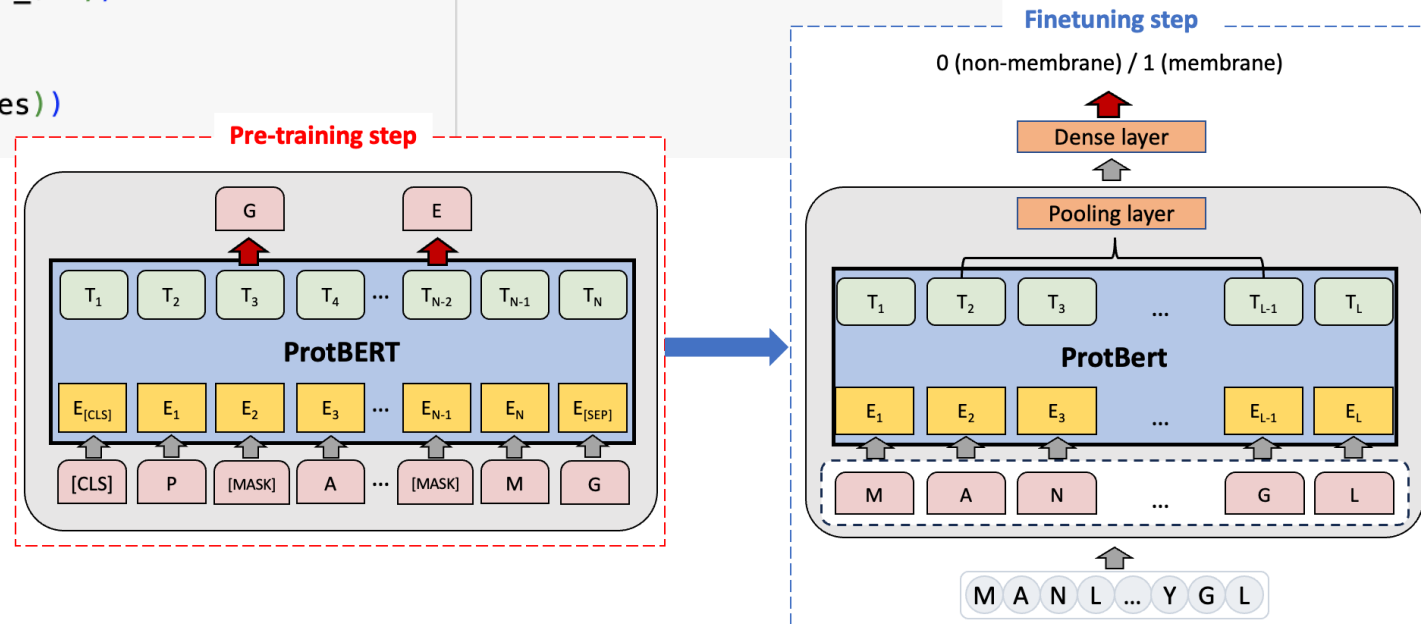


## Step 2. Build model for finetuning

- Protein sequence를 tokenization 하기
- Pretrained ProtBERT 불러오기
- 불러온 ProtBERT를 사용하여 amino-acid embedding 만들기
- Protein-level embedding을 위하여 amino-acid embedding들을 pooling하기
- **Dense layer 기반의 classifier 구성하여 예측하기**

```
13     layers = []
14     layers.append(nn.Linear(self.input_dim, hidden_dim))
15     layers.append(nn.ReLU())
16     layers.append(nn.Dropout(dropout))
17     for _ in range(num_layers - 2):
18         layers.append(nn.Linear(hidden_dim, hidden_dim))
19         layers.append(nn.ReLU())
20         layers.append(nn.Dropout(dropout))
21     layers.append(nn.Linear(hidden_dim, num_classes))
22     self.classifier = nn.Sequential(*layers)
```

```
37     logits = self.classifier(pooled_embeddings)
```



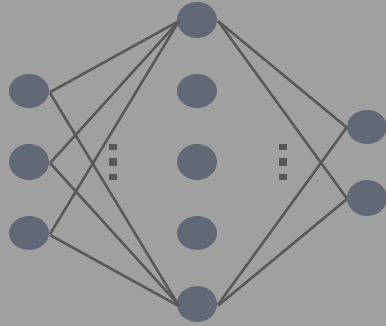
# Deep Learning Process

## Step1



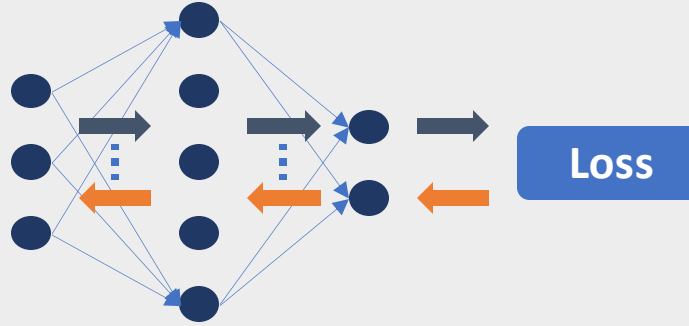
- Collecting data
- Preprocessing
- Dataset
- Data-loader

## Step2



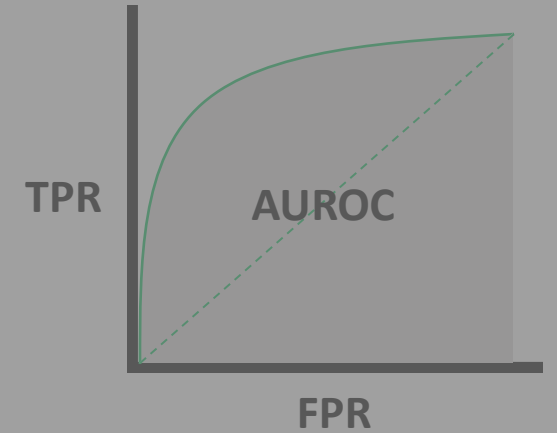
- Selecting method
- Building model

## Step3



- Hyper-parameters
- Loss function
- Optimizer
- Training

## Step4



- Predicting
- Evaluating

## Step 3. Training

---

```
1 # 주요 설정
2 device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
3
4 pretrained_model_name = "Rostlab/prot_bert"
5 num_classes = 2 # membrane or non-membrane
6 hidden_dim = 64 # 히든 레이어의 차원
7 num_layers = 3 # 레이어 수
8 batch_size = 64
9 epochs = 1 # 최대 에포크 수
10 learning_rate = 0.001
11 dropout = 0.0 # 드롭아웃 확률
```

```
1 # 모델 초기화
2 model = DeeplocPredictor(pretrained_model_name, hidden_dim, num_classes, device, num_layers=num_layers, dropout=dropout).to(device)
3 criterion = nn.CrossEntropyLoss()
4 optimizer = torch.optim.AdamW(filter(lambda p: p.requires_grad, model.parameters()), lr=learning_rate)
```



## Step 3. Training

---

- Epoch : 데이터셋 전체를 한 번 학습하는 주기
- Batch : 데이터셋을 나눈 작은 묶음
- Training loop는 각 epoch마다 모든 batch를 반복 학습하며, 각 batch마다 모델이 손실(loss)을 계산하고, 이를 통해 역전파 (backpropagation)로 가중치를 업데이트 하는 과정

```
for epoch in range(epochs):      # 사용자가 정의한 epoch 만큼 학습 진행
    model.train()                # model을 training mode로 변경
    :
    for batch in train_loader:   # batch 마다 모델 파라미터를 업데이트
        :
        outputs = model(batch)  # 모델에 batch 를 입력으로 넣어 output 계산
        loss = criterion(outputs, labels) # output과 label 사이의 오차(loss) 계산
        loss.backward()         # loss function 의 gradient 를 계산
        optimizer.step()        # 계산된 gradient 를 기반으로 하여 weight update 진행
        optimizer.zero_grad()   # 다음 step 을 위하여 gradient 초기화
    :
```

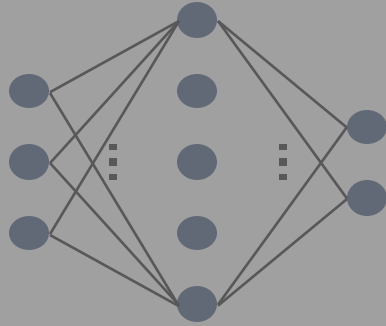
# Deep Learning Process

## Step1



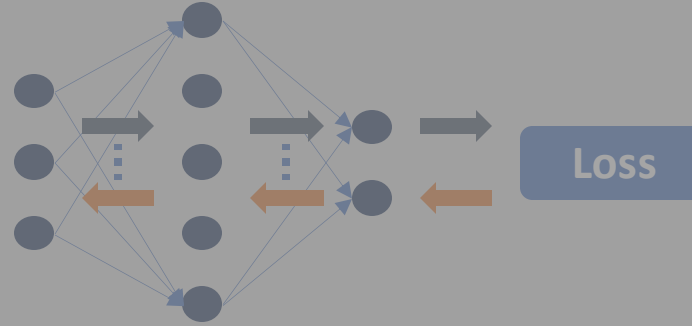
- Collecting data
- Preprocessing
- Dataset
- Data-loader

## Step2



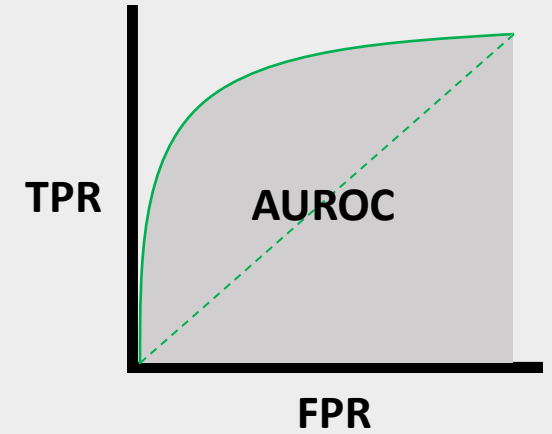
- Selecting method
- Building model

## Step3



- Hyper-parameters
- Loss function
- Optimizer
- Training

## Step4



- Predicting
- Evaluating



## Step 4. Evaluation

---

- Validation 단계에서는 모델의 성능을 평가하는 과정이므로 가중치 업데이트를 진행하지 않음
- `model.eval()`을 호출하여 모델을 평가모드로 전환
- With `torch.no_grad()`를 사용하여 미분 계산을 비활성화하여 메모리 사용을 줄이고, 계산 속도를 높임

```
for epoch in range(epochs):           # 사용자가 정의한 epoch 만큼 학습 진행
    model.eval()                       # model을 evaluation mode로 변경
    :
    with torch.no_grad():              # gradient 계산 비활성화
        for batch in valid_loader:    # batch 마다 forward propagation 진행
            :
            outputs = model(batch)     # 모델에 batch 를 입력으로 넣어 output 계산
            loss = criterion(outputs, labels) # output과 label 사이의 오차 (loss) 계산
            :
```

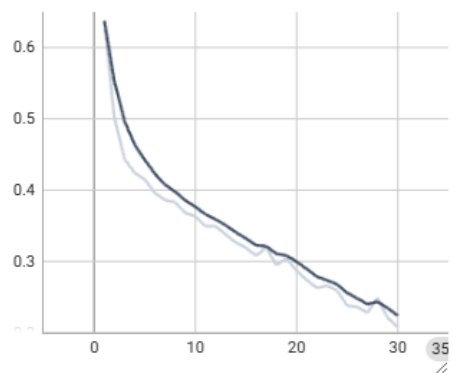
# Visualization

```
1 # TensorBoard writer 초기화
2 tensorboard_log_dir = "runs/deeploc_membrane_prediction"
3 os.makedirs(tensorboard_log_dir, exist_ok=True)
4 writer = SummaryWriter(tensorboard_log_dir)
```

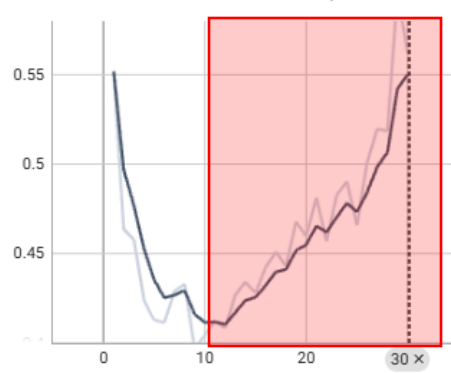
```
# TensorBoard에 Training 기록
writer.add_scalar('Loss/Train', avg_train_loss, epoch + 1)
writer.add_scalar('Accuracy/Train', train_accuracy, epoch + 1)
```

```
# TensorBoard에 Validation 기록
writer.add_scalar('Loss/Validation', avg_val_loss, epoch + 1)
writer.add_scalar('Accuracy/Validation', val_accuracy, epoch + 1)
```

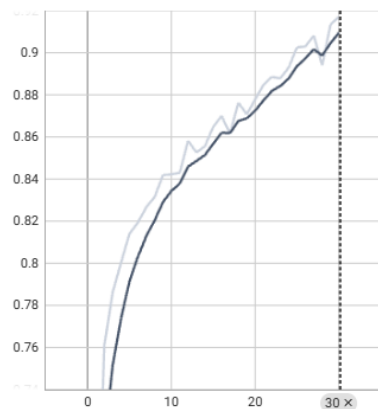
Loss/Train\_epoch



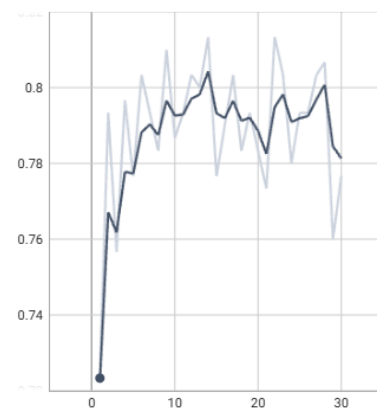
Loss/Validation\_epoch



Accuracy/Train\_epoch



Accuracy/Validation\_epoch

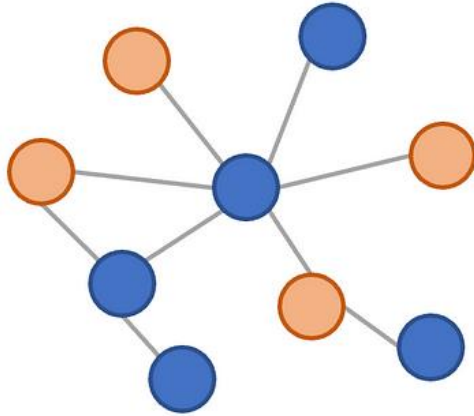


# Link Prediction

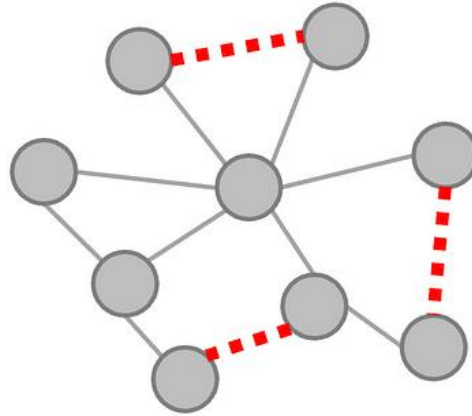


# Tasks of Graph

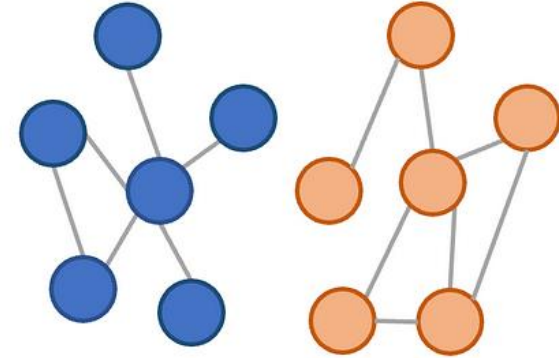
**Node Classification**



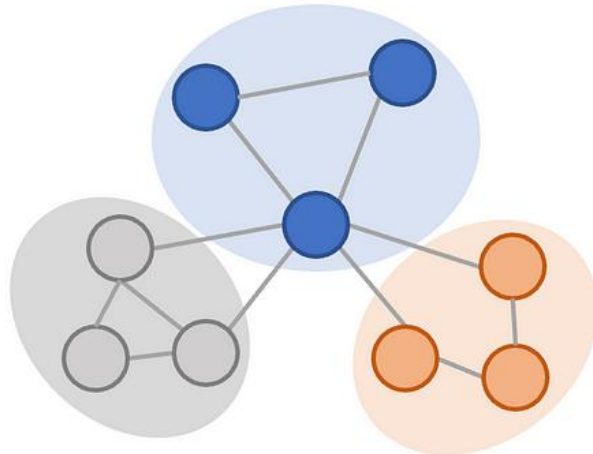
**Link Prediction**



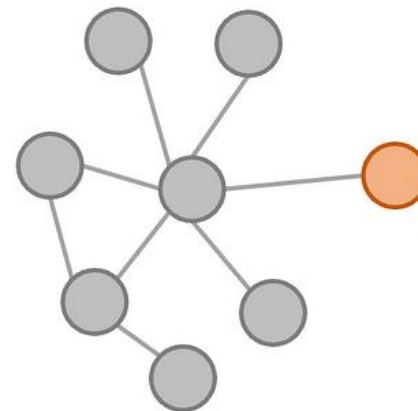
**Graph Classification**



**Community Detection**



**Anomaly Detection**



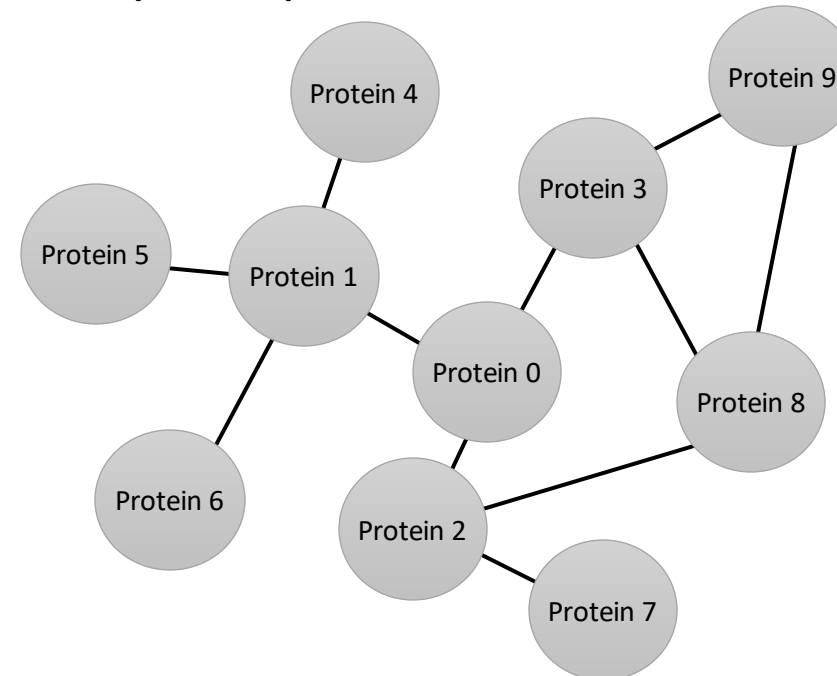
# Dataset

## PPI (Protein-Protein Interaction)

- This networks from the [“Predicting Multicellular Function through Multi-layer Tissue Networks”](#) paper
- Containing 50 features (positional gene sets, motif gene sets and immunological signatures, ...)
- [https://pytorch-geometric.readthedocs.io/en/stable/generated/torch\\_geometric.datasets.PPI.html#torch\\_geometric.datasets.PPI](https://pytorch-geometric.readthedocs.io/en/stable/generated/torch_geometric.datasets.PPI.html#torch_geometric.datasets.PPI)

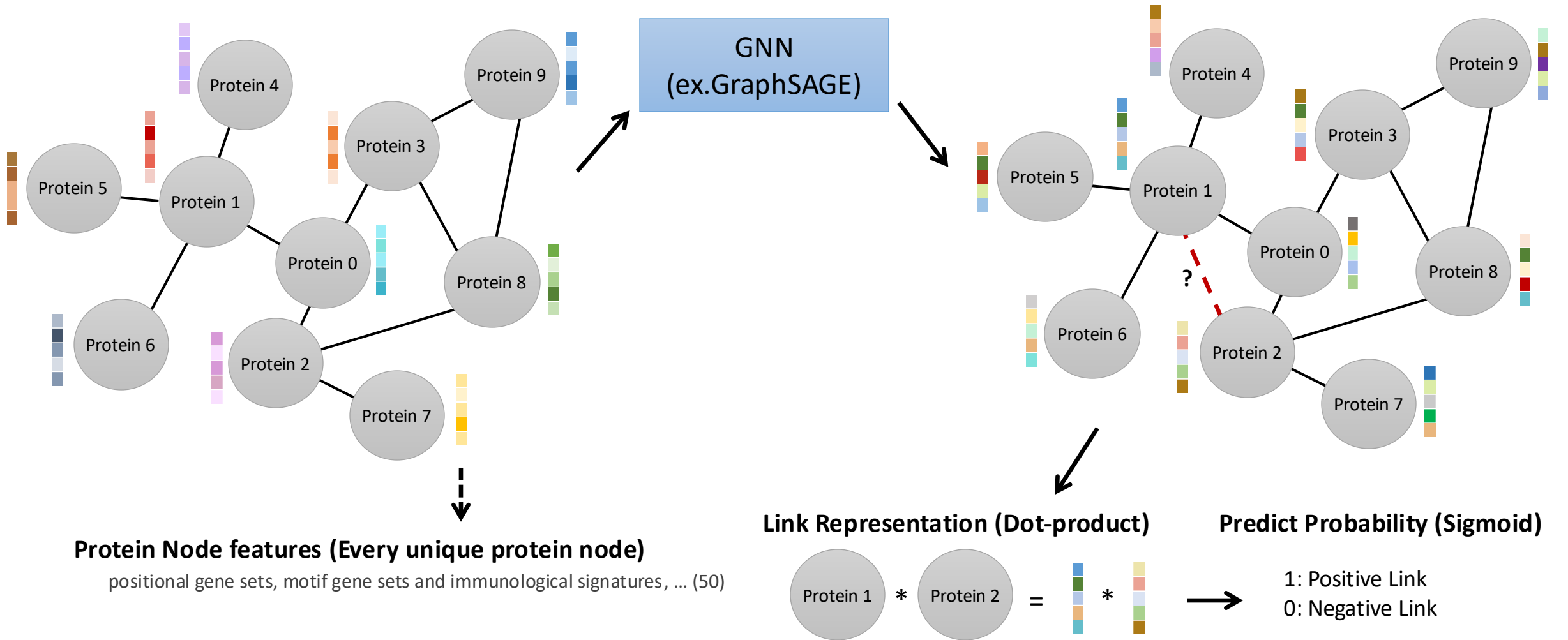
| #graphs | #nodes   | #edges    | #features |
|---------|----------|-----------|-----------|
| 20      | ~2,245.3 | ~61,318.4 | 50        |

PPI Graph Example



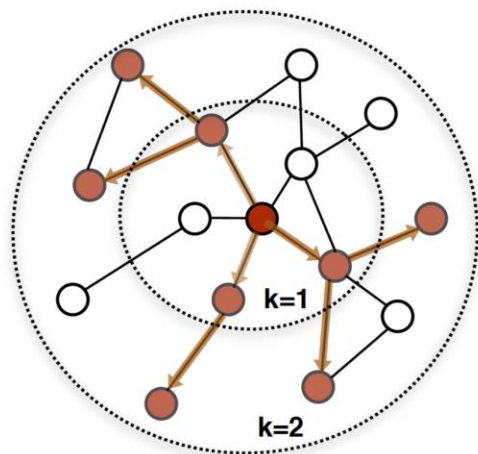
# Link Prediction

## Pipeline

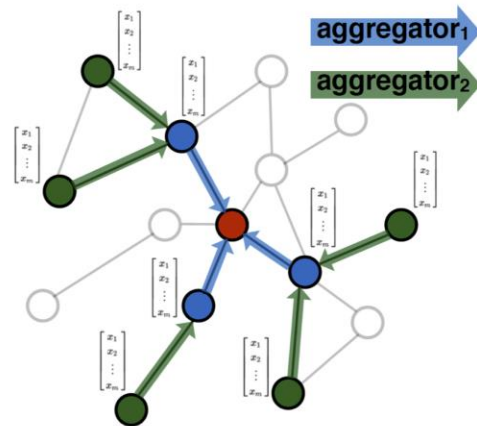


# GNN Model

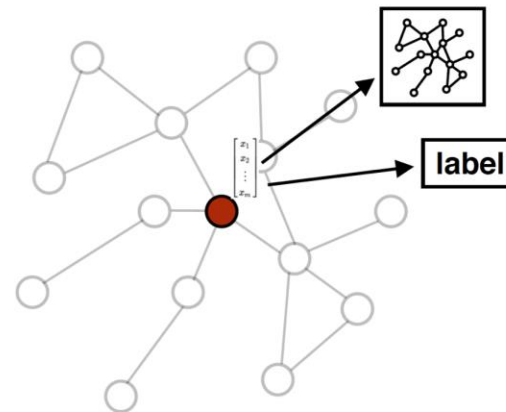
## GraphSAGE <https://arxiv.org/pdf/1706.02216>



1. Sample neighborhood



2. Aggregate feature information from neighbors



3. Predict graph context and label using aggregated information

### 1. 특정 단백질의 주변 이웃을 샘플링

사용자가 정의한 k-hop 단계의 이웃을 포함하여 정보를 추출

### 2. 이웃 특징 정보 집계

샘플링된 이웃의 특징 벡터를 학습 가능한 집계 함수(aggregator)를 사용하여 통합

Aggregator에는 Mean, LSTM, Pooling 등이 사용될 수 있음

### 3. 그래프 문맥과 노드 레이블 예측

집계된 정보를 바탕으로 단백질의 기능 또는 소속 그룹(노드 레이블)을 예측

전체 그래프의 문맥을 반영하여 단백질 간의 관계를 파악할 수 있음

# Environment

---

- **Python3**

- **Pytorch**

```
pip install torch
```

- **Pytorch Geometric**

```
pip install torch-geometric
```



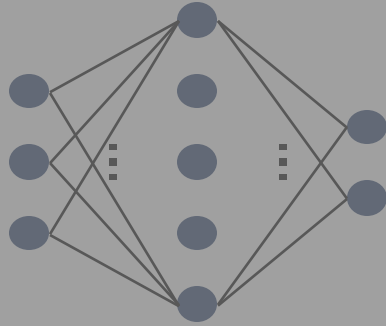
# Deep Learning Process

## Step1



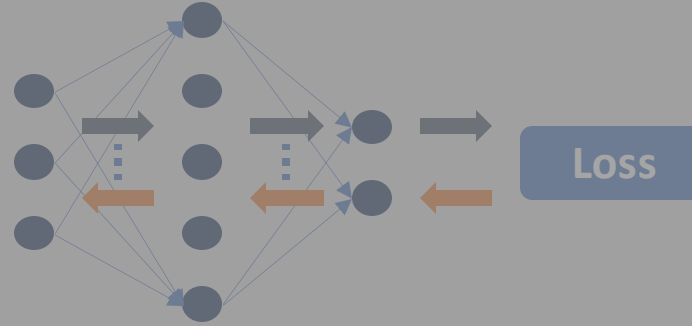
- Collecting data
- Preprocessing
- Dataset
- Data-loader

## Step2



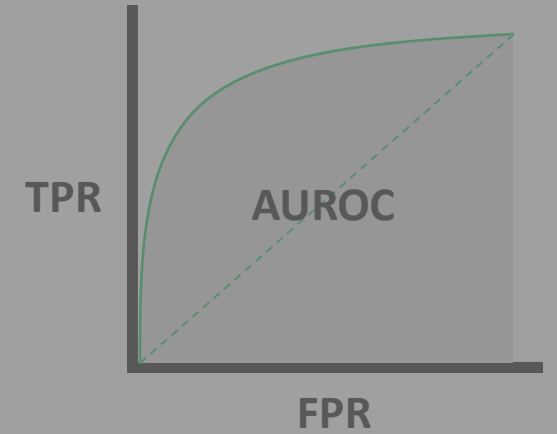
- Selecting method
- Building model

## Step3



- Hyper-parameters
- Loss function
- Optimizer
- Training

## Step4



- Predicting
- Evaluating

## Step 1. Load dataset

데이터셋 다운로드: from torch\_geometric.datasets import PPI  
ppi = PPI(root="./data/PPI")

첫 번째 서브 그래프 사용: data = ppi[0]

그래프의 노드 개수: data.num\_nodes → Number of nodes: 1767

그래프의 엣지 개수: data.edge\_index.size(1) → Number of edges: 32318

엣지 인덱스 차원: data.edge\_index.shape → Edges shape: torch.Size([2, 32318])

엣지 인덱스 구성: data.edge\_index → Edge index:  
tensor([[ 0, 0, 0, ..., 1744, 1745, 1749],  
[ 372, 1101, 766, ..., 1745, 1744, 1739]])

노드 특성 차원 확인 (50차원): data.x.shape → Node feature shape: torch.Size([1767, 50])

노드 특성 예시: data.x[0] → Node feature Example:  
tensor([-0.0855, -0.0884, -0.1128, -0.1719, -0.0766, -0.1003, -0.0751, -0.1149,  
-0.1212, -0.0994, 0.0000, -0.1699, -0.0428, -0.1123, -0.0760, -0.1152,  
-0.1031, -0.1120, -0.1435, -0.0975, -0.0875, -0.1457, -0.1234, -0.1242,  
-0.0976, -0.1197, -0.1161, -0.0735, -0.0667, -0.0873, -0.1797, -0.1447,  
-0.1606, -0.1582, -0.1477, -0.4350, -0.1617, -0.1556, -0.1526, -0.1396,  
-0.1281, -0.1539, -0.1593, -0.1546, -0.1466, -0.1449, -0.1568, -0.1399,  
-0.1494, -0.1481])

그래프 전체 구성: data → Data(x=[1767, 50], edge\_index=[2, 32318])

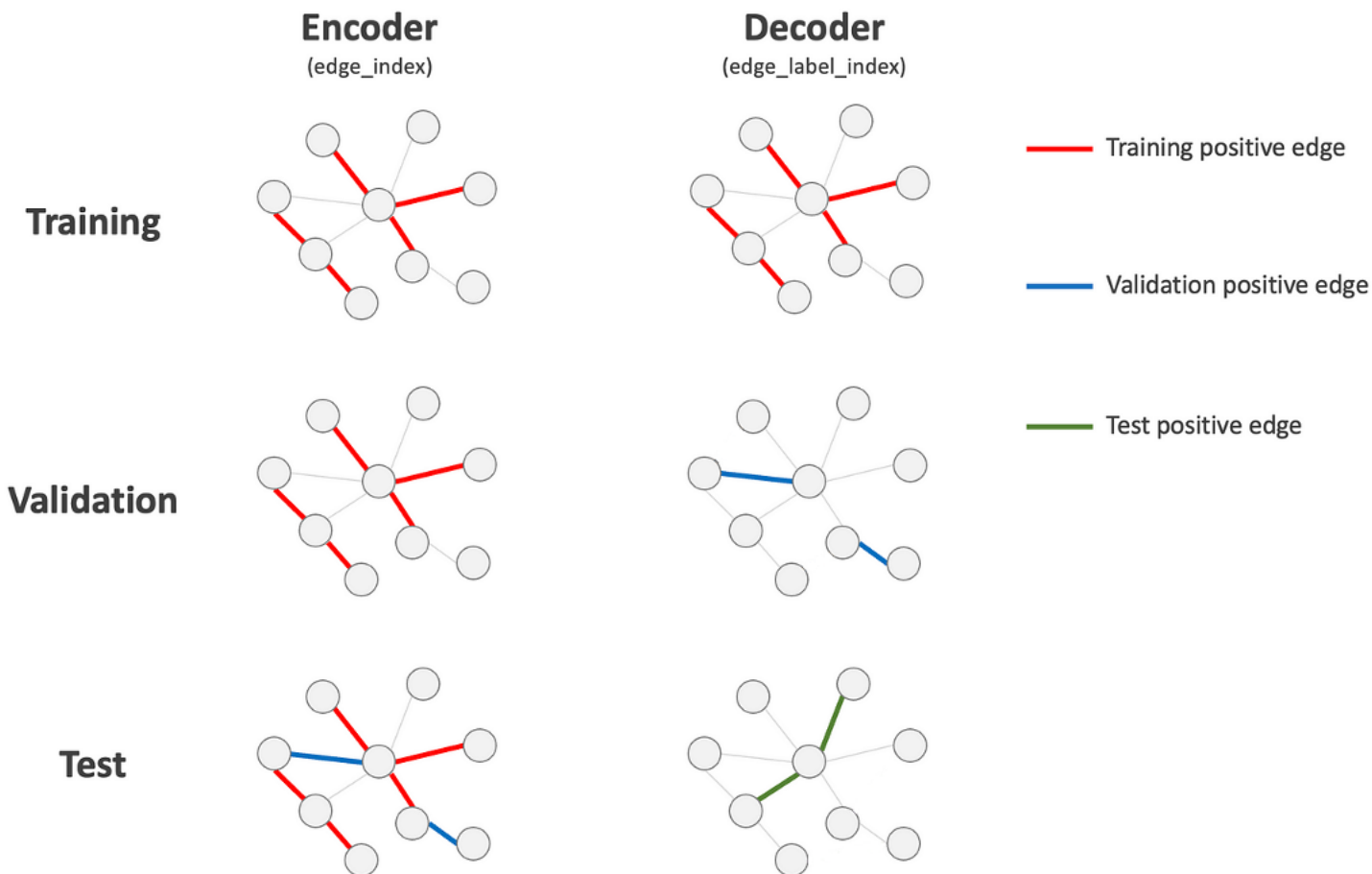
# Step 1. Load dataset

```
from torch_geometric.transforms import RandomLinkSplit

# 데이터셋 분리 (학습, 검증, 테스트)
# RandomLinkSplit은 데이터를 학습, 검증, 테스트 세트로 나눕니다.
# 엣지의 70%를 학습용, 10%를 검증용, 20%를 테스트용으로 사용합니다.
transform = RandomLinkSplit(
    num_val=0.1,
    num_test=0.2,
    add_negative_train_samples=False,
    disjoint_train_ratio=0.3,
    neg_sampling_ratio=0)

train_data, val_data, test_data = transform(data)
print("Train: ", train_data)
print("Validation: ", val_data)
print("Test: ", test_data)
```

Encoder: Graph 구조 학습을 위한 Link  
Decoder: Supervision을 위해 사용하는 Link



```
Train: Data(x=[1767, 50], edge_index=[2, 15837], edge_label=[6787], edge_label_index=[2, 6787])
Validation: Data(x=[1767, 50], edge_index=[2, 22624], edge_label=[3231], edge_label_index=[2, 3231])
Test: Data(x=[1767, 50], edge_index=[2, 25855], edge_label=[6463], edge_label_index=[2, 6463])
```

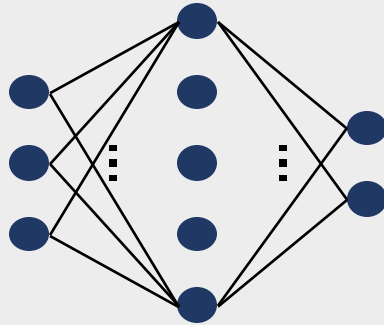
# Deep Learning Process

## Step1



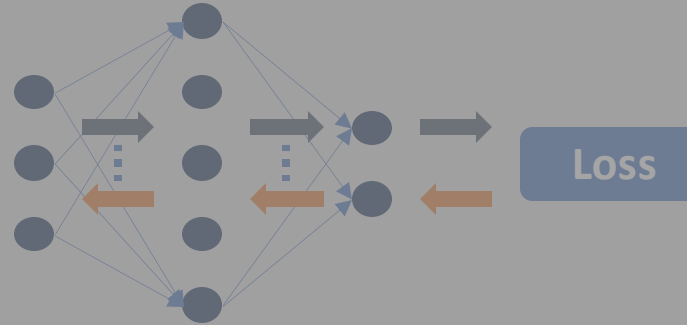
- Collecting data
- Preprocessing
- Dataset
- Data-loader

## Step2



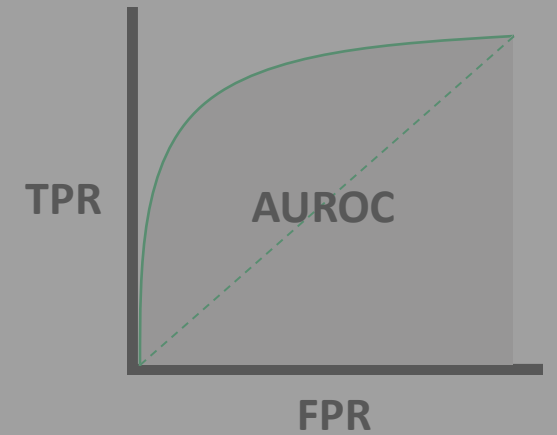
- Selecting method
- Building model

## Step3



- Hyper-parameters
- Loss function
- Optimizer
- Training

## Step4



- Predicting
- Evaluating

## Step 2. Build model

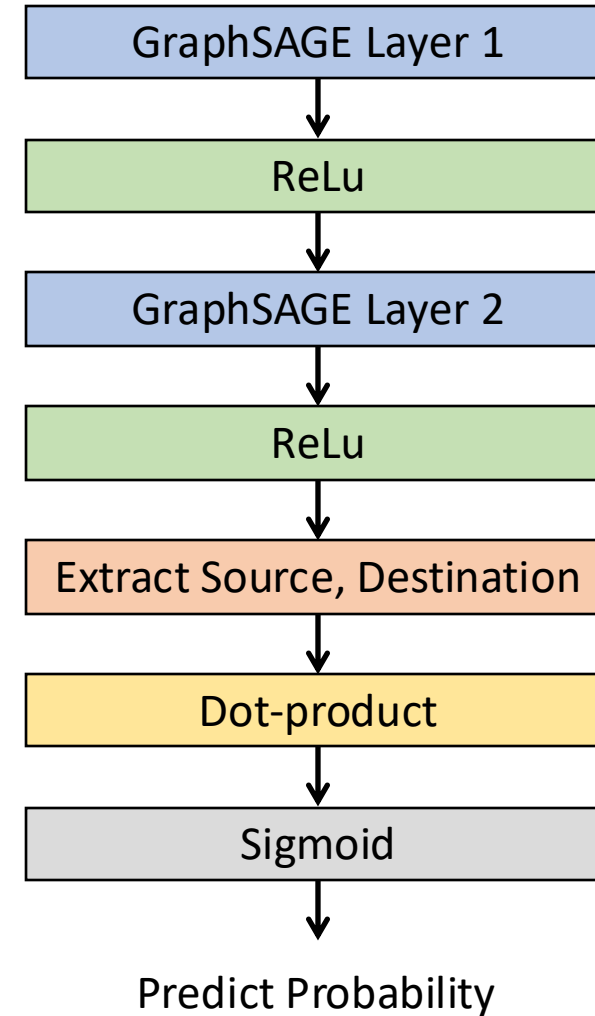
```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch_geometric.nn import SAGEConv

# 모델 정의
# GraphSAGE 레이어를 사용하는 모델 정의
class GNNForLinkPrediction(nn.Module):
    def __init__(self, in_channels, hidden_channels):
        super().__init__()
        # SAGEConv 레이어 정의
        self.conv1 = SAGEConv((-1, -1), hidden_channels) # 첫 번째 GraphSAGE 레이어
        self.conv2 = SAGEConv(hidden_channels, hidden_channels) # 두 번째 GraphSAGE 레이어

    def forward(self, x, edge_index, edge_label_index):
        # 첫 번째 GraphSAGE 레이어를 통과시키고 ReLU 활성화 함수 적용
        x = F.relu(self.conv1(x, edge_index))
        # 두 번째 GraphSAGE 레이어를 통과
        x = F.relu(self.conv2(x, edge_index))
        src, dst = edge_label_index

        # 소스 및 타겟 노드의 임베딩 추출 및 점곱 계산
        src, dst = edge_label_index # 엣지의 소스와 타겟 노드 인덱스
        edge_pred = torch.sigmoid((x[src] * x[dst]).sum(dim=-1)) # dotproduct로 스칼라 값 출력
        return edge_pred

# 모델 초기화
model = GNNForLinkPrediction(
    in_channels=train_data.num_features,
    hidden_channels=64 # 은닉층의 차원
)
```



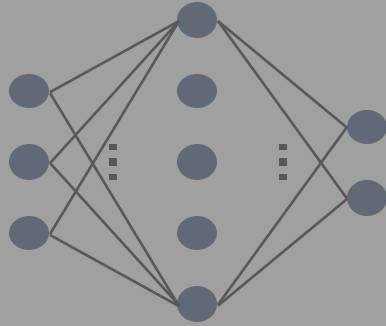
# Deep Learning Process

## Step1



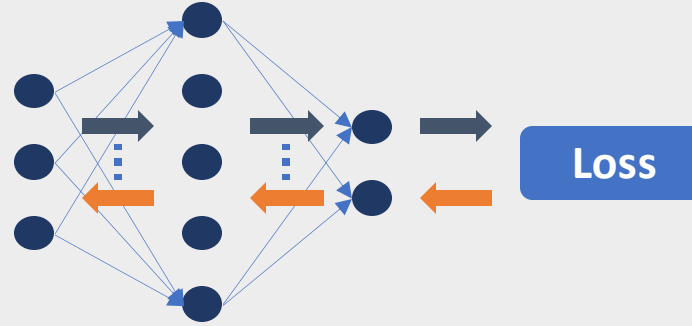
- Collecting data
- Preprocessing
- Dataset
- Data-loader

## Step2



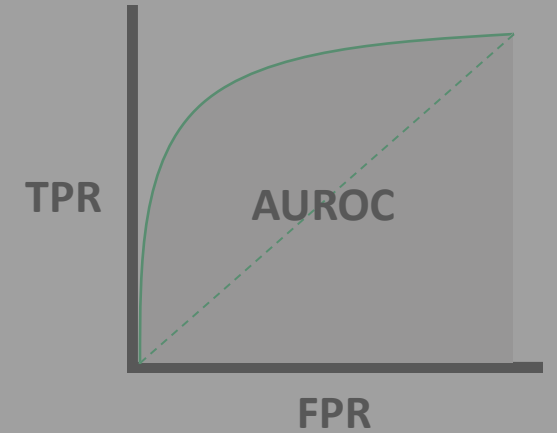
- Selecting method
- Building model

## Step3



- Hyper-parameters
- Loss function
- Optimizer
- Training

## Step4



- Predicting
- Evaluating

## Step 3. Training

Batch 생성을 위한 Subgraph 생성하는 Loader 구성

사용하는 Data(Train, Validation, Test)에 대해 각각 모두 구성 필요

[torch geometric의 LinkNeighborLoader Option]

- batch\_size : Batch 크기
- edge\_label\_index : Supervision에 사용하는 엣지 인덱스
- num\_neighbors : 각 Layer에서 샘플링할 이웃 노드 수
- neg\_sampling\_ratio : negative edge를 생성할 비율 (positive edge 대비)
- shuffle : Data shuffle 여부

Training의 경우에만, 특정 패턴 학습에 대한 overfitting 방지를 위해 True

```
from torch_geometric.loader import LinkNeighborLoader
# 데이터 로더 설정
# LinkNeighborLoader는 이웃 노드 샘플링을 통해 배치를 생성합니다.
train_loader = LinkNeighborLoader(
    train_data,
    batch_size=32, # 배치 크기
    edge_label_index=train_data.edge_label_index, # 학습용 엣지 레이블 인덱스
    num_neighbors=[10, 10], # 각 레이어에서 샘플링할 이웃 노드 수
    neg_sampling_ratio=1.0, # negative edge를 positive edge 수의 1배로 샘플링
    shuffle=True
)

val_loader = LinkNeighborLoader(
    val_data,
    batch_size=32,
    edge_label_index=val_data.edge_label_index,
    num_neighbors=[10, 10],
    neg_sampling_ratio=1.0,
    shuffle=False
)

test_loader = LinkNeighborLoader(
    test_data,
    batch_size=32,
    edge_label_index=test_data.edge_label_index,
    num_neighbors=[10, 10],
    neg_sampling_ratio=1.0,
    shuffle=False
)
```

## Step 3. Training

Training loop는 각 epoch마다 모든 batch (앞서 생성한 subgraph)를 반복 학습하며  
각 Batch 마다 모델이 손실(loss)를 계산하며, 이를 역전파(Backpropagation)로 가중치를 업데이트

```
# 학습 및 검증
for epoch in range(10):
    print(f"\nEpoch {epoch+1}:")
    train_loss = train() # 학습 수행
    val_loss, val_auc, _, _ = evaluate(val_loader, epoch, mode="Validation")
    print(f"Train Loss: {train_loss:.4f}, Validation Loss: {val_loss:.4f}")
```

```
# 학습 루프 정의
def train():
    model.train() # 모델을 학습 모드로 설정
    total_loss = 0
    loop = tqdm(train_loader, desc="Training") # tqdm으로 학습 상태 표시
    for batch in loop:
        optimizer.zero_grad() # 옵티마이저의 그래디언트 초기화
        edge_pred = model(batch.x, batch.edge_index, batch.edge_label_index) # 모델 출력 계산
        loss = criterion(edge_pred, batch.edge_label.float()) # 손실 계산
        loss.backward() # 역전파 단계
        optimizer.step() # 가중치 업데이트

        total_loss += loss.item()
        loop.set_postfix(loss=loss.item()) # tqdm 진행 상태에 현재 손실 표시
    avg_loss = total_loss / len(train_loader)
    writer.add_scalar("Loss/Train", avg_loss, epoch) # TensorBoard 기록
    return avg_loss
```



## Step 3. Training

Validation 단계에서는 모델의 성능을 평가하는 과정으로, 가중치 업데이트를 진행하지 않음

model.eval()을 통해 모델을 평가 모드로 설정

with torch.\_no\_grad()를 사용하여 미분 계산을 비활성화하여 메모리 사용을 줄이고 계산 속도를 높임

```
# 평가 함수 정의
def evaluate(loader, epoch, mode="Validation"):
    model.eval() # 모델을 평가 모드로 설정
    preds, labels = [], []
    loop = tqdm(loader, desc="Evaluating") # tqdm으로 평가 상태 표시
    with torch.no_grad(): # 그래디언트 계산 비활성화
        for batch in loop:
            edge_pred = model(batch.x, batch.edge_index, batch.edge_label_index) # 모델 출력 계산
            preds.append(edge_pred.cpu()) # 예측값 저장
            labels.append(batch.edge_label.cpu()) # 실제 라벨 저장
    preds, labels = torch.cat(preds), torch.cat(labels)
    loss = criterion(preds, labels.float()).item()
    auc = roc_auc_score(labels, preds)
    writer.add_scalar(f"Loss/{mode}", loss, epoch) # TensorBoard 기록
    writer.add_scalar(f"AUC/{mode}", auc, epoch)
    return loss, auc, preds, labels
```

Epoch 1:

Training: 100%|██████████| 213/213 [00:03<00:00, 68.09it/s, loss=0.81]

Evaluating: 100%|██████████| 101/101 [00:00<00:00, 180.07it/s]

Train Loss: 0.6756, Validation Loss: 0.6614

...

Epoch 10:

Training: 100%|██████████| 213/213 [00:03<00:00, 63.33it/s, loss=0.462]

Evaluating: 100%|██████████| 101/101 [00:00<00:00, 175.79it/s]

Train Loss: 0.5936, Validation Loss: 0.6256

→ Validation Loss도 함께 확인하여 모델 Overfitting 정도 확인

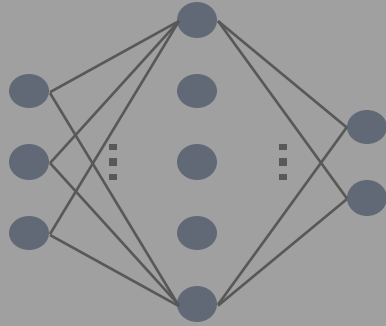
# Deep Learning Process

## Step1



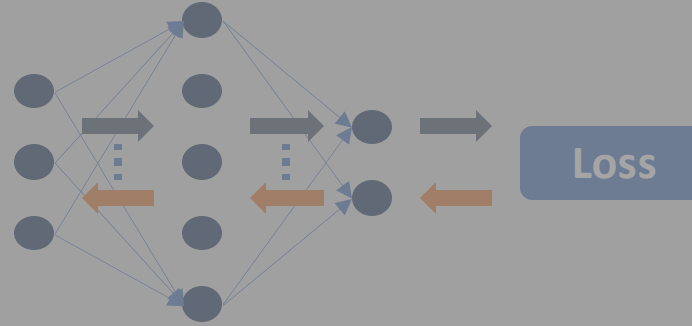
- Collecting data
- Preprocessing
- Dataset
- Data-loader

## Step2



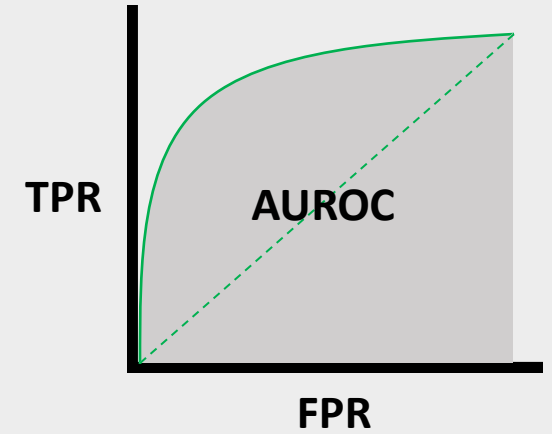
- Selecting method
- Building model

## Step3



- Hyper-parameters
- Loss function
- Optimizer
- Training

## Step4



- Predicting
- Evaluating

## Step 4. Evaluation

---

모델 학습이 모두 완료된 후, 테스트 데이터에 대한 모델 성능 평가 진행

평가 방식: AUC-ROC

```
# 테스트
# 테스트 데이터에서 모델 성능 평가
print("\nEvaluating Test Set")
test_loss, test_auc, test_preds, test_labels = evaluate(test_loader, 10, mode="Test")
print(f"\nTest Loss: {test_loss:.4f}, Test AUC: {test_auc:.4f}")
```

Evaluating Test Set

Evaluating: 100%|██████████| 202/202 [00:01<00:00, 201.23it/s]

Test Loss: 0.6222, Test AUC: 0.8244

## Step 4. Evaluation

테스트 데이터에서 실제 예측이 어떻게 이루어졌는지 5개의 예시 확인

```
# 테스트 데이터에서 예측 예시 출력
def print_test_examples(num_examples=5):
    #테스트 데이터에서 예측값, 실제값, 원래 엣지 정보를 출력합니다.
    for i in range(num_examples): # num_examples만큼 출력
        # 테스트 데이터의 원래 엣지 인덱스 추출
        test_edge_index = test_data.edge_label_index[:, i] # 원래 엣지의 (src, dst) 인덱스
        test_edge_src, test_edge_dst = test_edge_index[0].item(), test_edge_index[1].item()

        # 모델의 예측값과 실제 라벨
        test_example_pred = test_preds[i].item()
        test_example_label = test_labels[i].item()
        predicted_label = 1 if test_example_pred > 0.5 else 0 # 예측 라벨 계산 threshold 0.5

        # 결과 출력
        print(f"Test Edge {i + 1}: (src: {test_edge_src}, dst: {test_edge_dst})")
        print(f"Prediction Probability: {test_example_pred:.4f}, Predicted Label: {predicted_label}, Ground Truth: {test_example_label}\n")

# 테스트 예시 출력
print_test_examples(num_examples=5) # 상위 5개 예시 출력
```

Test Edge 1: (src: 1518, dst: 498)

Prediction Probability: 0.8915, Predicted Label: 1, Ground Truth: 1.0

Test Edge 2: (src: 1053, dst: 496)

Prediction Probability: 0.9009, Predicted Label: 1, Ground Truth: 1.0

Test Edge 3: (src: 425, dst: 366)

Prediction Probability: 0.5270, Predicted Label: 1, Ground Truth: 1.0

Test Edge 4: (src: 311, dst: 646)

Prediction Probability: 0.8610, Predicted Label: 1, Ground Truth: 1.0

Test Edge 5: (src: 1497, dst: 890)

Prediction Probability: 0.7808, Predicted Label: 1, Ground Truth: 1.0

→ 1518번 단백질과 498번 단백질은 모델의 예측 확률이 0.8915임으로 threshold 0.5에 따라 관계가 존재하는 것으로 예측했으며, 실제 정답도 관계가 존재하는 것으로 나타남

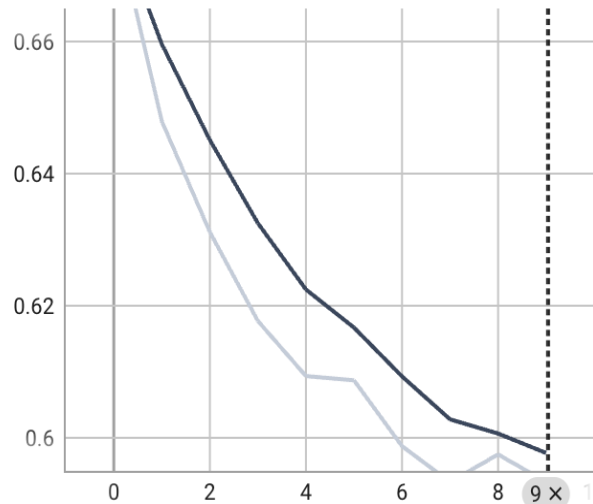


# Visualization

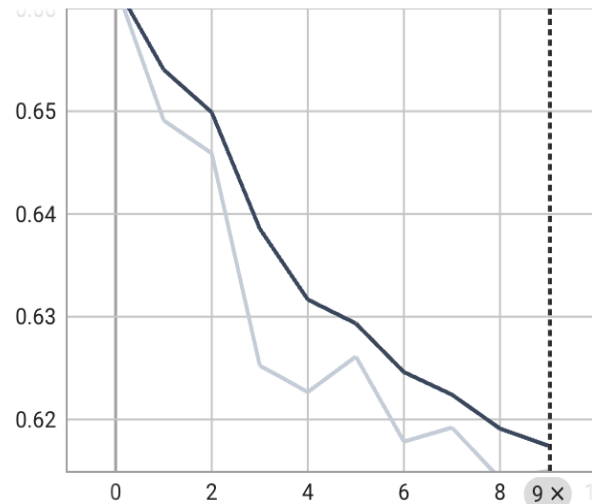
```
# TensorBoard
print("To visualize results in Colab, run:")
print("%load_ext tensorboard")
print(f"%tensorboard --logdir {log_dir}")
writer.close()
```

```
%load_ext tensorboard
%tensorboard --logdir /content/runs/ppi_link_prediction
```

Loss/Train



Loss/Validation



AUC/Validation

