Intro Deep Learning - Part 3

What's the goal for today?

- Start learning and practice Pytorch
- Learn more building blocks for neural networks

Intro Deep Learning - Part 3

What's the goal for today?

- Start learning and practice Pytorch
- Learn more building blocks for neural networks

What are we doing ?

- MLP
- Auto-encoders
- Achieve > 90% accuracy on MNIST
- Regularization Techniques

Intro Deep Learning - Part 3

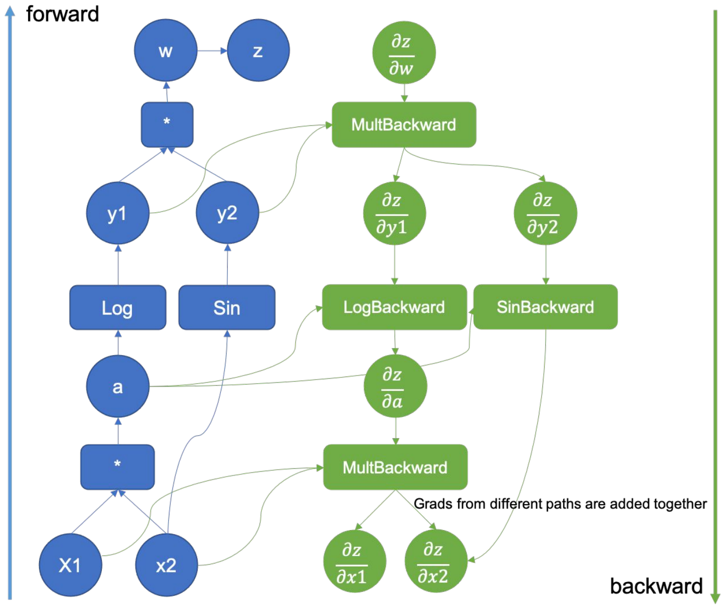
Pytorch (https://pytorch.org/)

- Deep Learning Framework by Meta
- Auto Differentiation
- Python Interface
- Open Source
- Widely Used

Intro Deep Learning - Part 3

Pytorch (https://pytorch.org/)

- Deep Learning Framework by Meta
- Auto Differentiation
- Python Interface
- Open Source
- Widely Used



https://pytorch.org/blog/computational-graphs-constructed-in-pytorch/

Intro Deep Learning - Part 3

Pytorch (https://pytorch.org/)

- Deep Learning Framework by Meta
- Auto Differentiation
- Python Interface
- Open Source
- Widely Used

- Tensors
- Datasets
- DataLoaders
- Transforms
- Build Model
- Automatic Differentiation
- Optimization Loop
- Save, Load, Use Model

Intro Deep Learning - Part 3

Pyt

```python
import torch

# Creating a tensor
x = torch.tensor([1, 2, 3])

# Element-wise operations
y = x + 2
print(y)
```

Intro Deep Learning - Part 3

Pyt

```python
import torch

# Creating a tensor
x = torch.tensor([1, 2, 3])

# Element-wise operations
y = x + 2
print(y)
```

```python
import torch

# Creating a tensor with gradient tracking
x = torch.tensor([2.0], requires_grad=True)

# Define a function
y = x**2

# Compute gradients
y.backward()

print(x.grad)  # Should print 4.0
```

Intro Deep Learning - Part 3

```python
import torch
import torch.nn as nn
import torch.optim as optim

# Create a dataset
X = torch.rand(100, 1)
y = 3 * X + 2 + 0.1 * torch.rand(100, 1)

# Define a neural network
model = nn.Linear(1, 1)

# Define a loss function and optimizer
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)
```

```python
# Training loop
for epoch in range(100):
    # Forward pass
    outputs = model(X)
    loss = criterion(outputs, y)

    # Backpropagation and optimization
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

# Get the trained model's parameters
trained_weight, trained_bias = model.parameters()
print("Trained Weight:", trained_weight)
print("Trained Bias:", trained_bias)
```

Intro Deep Learning - Part 3

F

```python
import torch
import torch.nn as nn
import torch.optim as optim

# Create a dataset
X = torch.rand(100, 1)
y = 3 * X + 2 + 0.1 * torch.rand(100, 1)

# Define a neural network
model = nn.Linear(1, 1)

# Define a loss function and optimizer
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)
```

```python
# Training loop
for epoch in range(100):
    # Forward pass
    outputs = model(X)
    loss = criterion(outputs, y)

    # Backpropagation and optimization
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

# Get the trained model's parameters
trained_weight, trained_bias = model.parameters()
print("Trained Weight:", trained_weight)
print("Trained Bias:", trained_bias)
```

Intro Deep Learning - Part 3

F

Ensure gradients are the only ones you need

```python
y = 3 * X + 2 + 0.1 * torch.rand(100, 1)

# Define a neural network
model = nn.Linear(1, 1)

# Define a loss function and optimizer
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)
```

```python
# Training loop
for epoch in range(100):
    # Forward pass
    outputs = model(X)
    loss = criterion(outputs, y)

    # Backpropagation and optimization
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

# Get the trained model's parameters
trained_weight, trained_bias = model.parameters()
print("Trained Weight:", trained_weight)
print("Trained Bias:", trained_bias)
```

Intro Deep Learning - Part 3

Pyt

```
import torch
import torch.nn as nn

class BinaryClassifier(nn.Module):
    def __init__(self, input_size):
        super(BinaryClassifier, self).__init__()

        # Define the layers of the neural network
        self.fc1 = nn.Linear(input_size, 64)
        self.fc2 = nn.Linear(64, 32)
        self.fc3 = nn.Linear(32, 1)  # Output layer with a single unit for binary classification

        self.sigmoid = nn.Sigmoid()  # Sigmoid activation for binary classification

    def forward(self, x):
        x = torch.relu(self.fc1(x))  # ReLU activation for the first layer
        x = torch.relu(self.fc2(x))  # ReLU activation for the second layer
        x = self.fc3(x)  # Output layer

        output = self.sigmoid(x)  # Sigmoid activation for binary classification

        return output
```

Intro Deep Learning - Part 3

Pyt

```python
import torch
import torch.nn as nn

class BinaryClassifier(nn.Module):
    def __init__(self, input_size):
        super(BinaryClassifier, self).__init__()

        # Define the layers of the neural networ
        self.fc1 = nn.Linear(input_size, 64)
        self.fc2 = nn.Linear(64, 32)
        self.fc3 = nn.Linear(32, 1)  # Output la

        self.sigmoid = nn.Sigmoid()  # Sigmoid a

    def forward(self, x):
        x = torch.relu(self.fc1(x))  # ReLU acti
        x = torch.relu(self.fc2(x))  # ReLU acti
        x = self.fc3(x)  # Output layer

        output = self.sigmoid(x)  # Sigmoid acti

        return output
```

```python
# Usage example
input_size = 10  # Size of input features
model = BinaryClassifier(input_size)

# To use the model for prediction, you can pass input data through it:
input_data = torch.randn(1, input_size)  # Example input data
output = model(input_data)
print(output)
```

Intro Deep Learning - Part 3

Pyt

```python
import torch
import torch.nn as nn

class BinaryClassifier(nn.Module):
    def __init__(self, input_size):
        super(BinaryClassifier, self).__init__()

        # Define the layers of the neural networ
        self.fc1 = nn.Linear(input_size, 64)
        self.fc2 = nn.Linear(64, 32)
        self.fc3 = nn.Linear(32, 1)  # Output la

        self.sigmoid = nn.Sigmoid()  # Sigmoid a

    def forward(self, x):
        x = torch.relu(self.fc1(x))  # ReLU acti
        x = torch.relu(self.fc2(x))  # ReLU acti
        x = self.fc3(x)  # Output layer

        output = self.sigmoid(x)  # Sigmoid acti

        return output
```

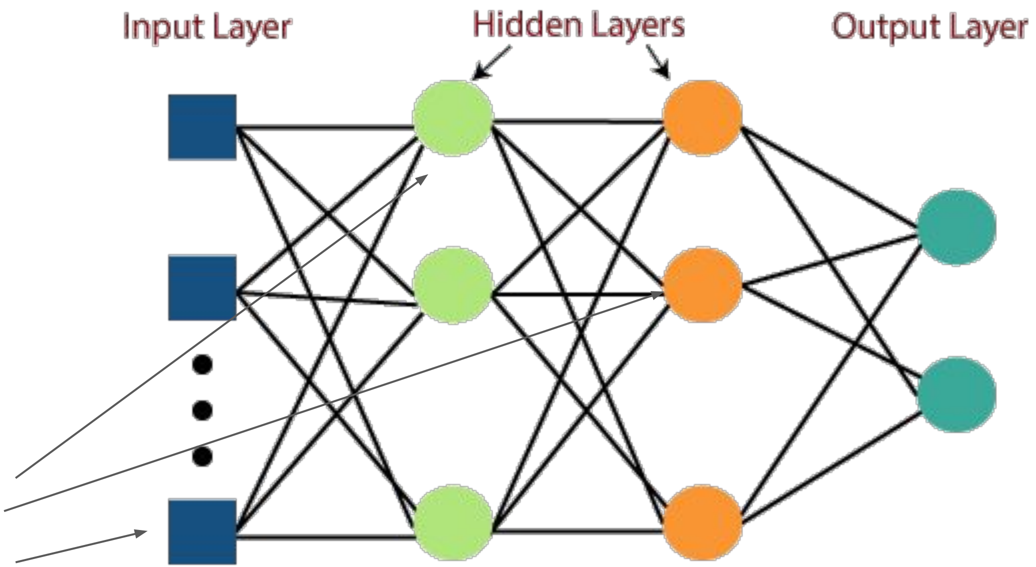
```python
# Usage example
input_size = 10  # Size of input features
model = BinaryClassifier(input_size)

# To use the model for prediction, you can pass input data through it:
input_data = torch.randn(1, input_size)  # Example input data
output = model(input_data)
print(output)
```

Intro Deep Learning - Part 3

Pytorch (https://pytorch.org/)

- Deep Learning Framework by Meta
- Auto Differentiation
- Python Interface
- Open Source
- Widely Used

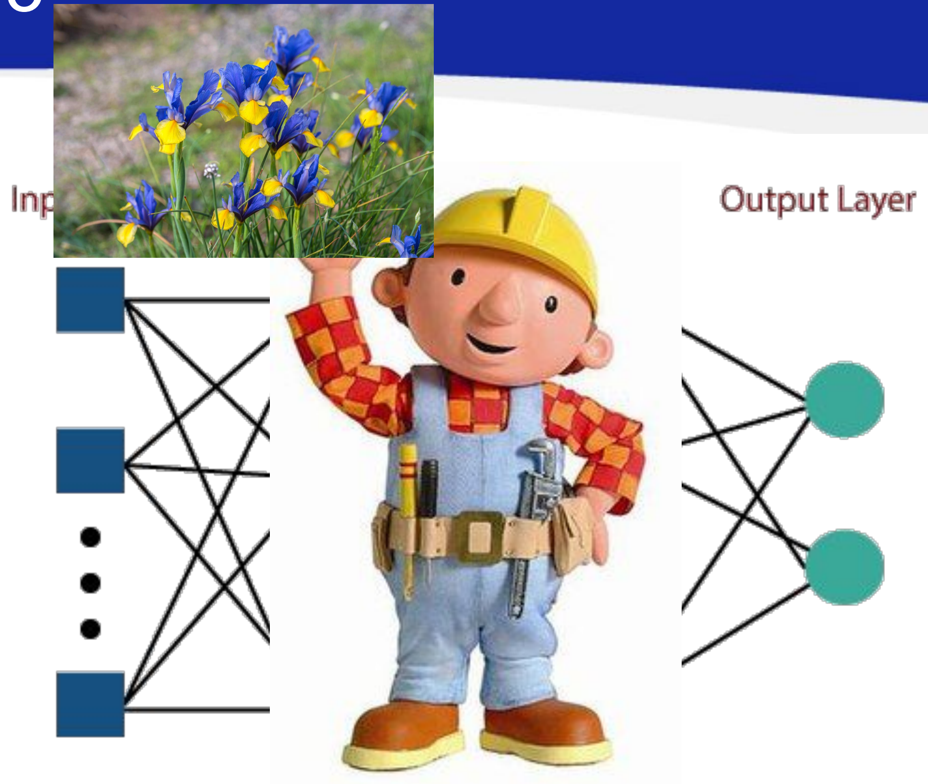
```
nn.Linear
```



Intro Deep Learning - Part 3

Pytorch (https://pytorch.org/)

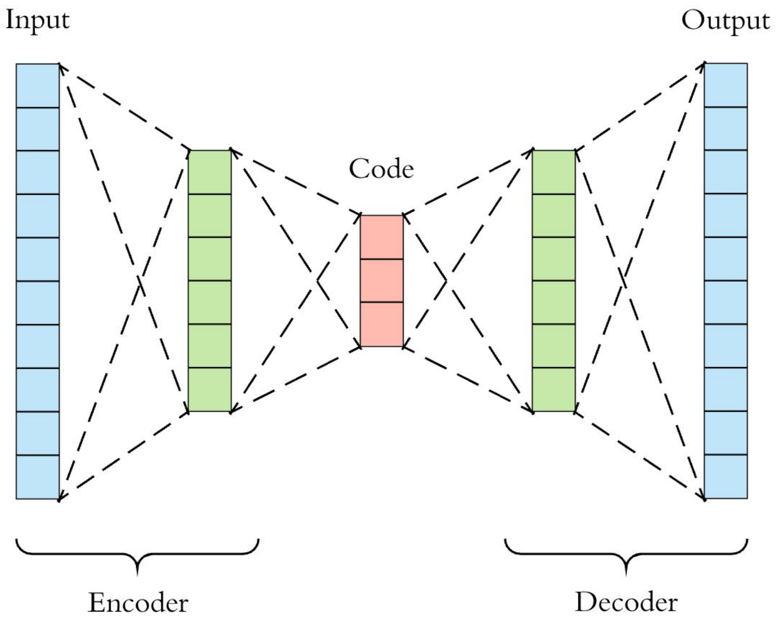
- Deep Learning Framework by Meta
- Auto Differentiation
- Python Interface
- Open Source
- Widely Used



Intro Deep Learning - Part 3

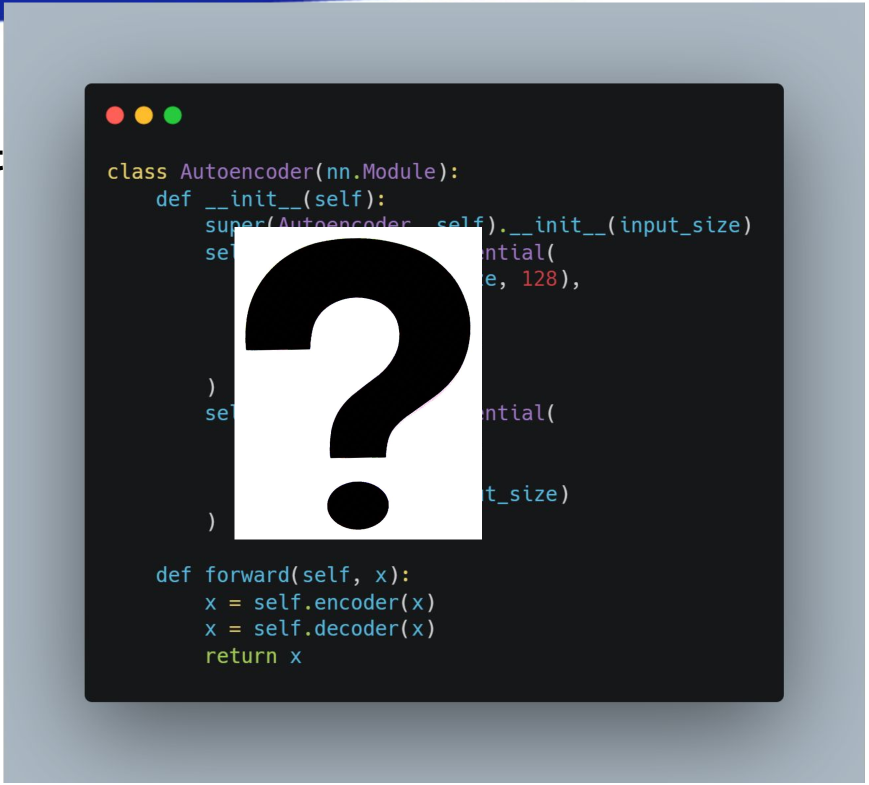
Auto-encoders

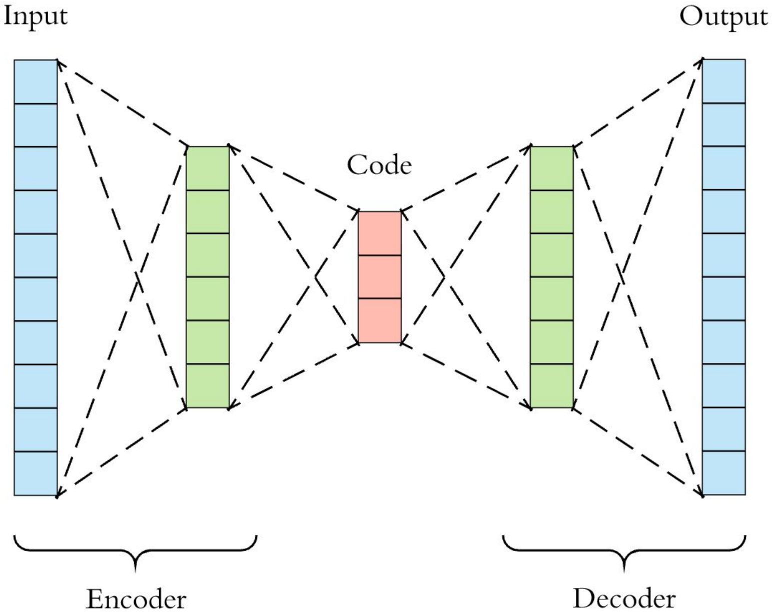
- Learns features
- Data compression
- Reduces dimensions
- Unsupervised Learning approach



Intro Deep Learning - Part 3

Aut



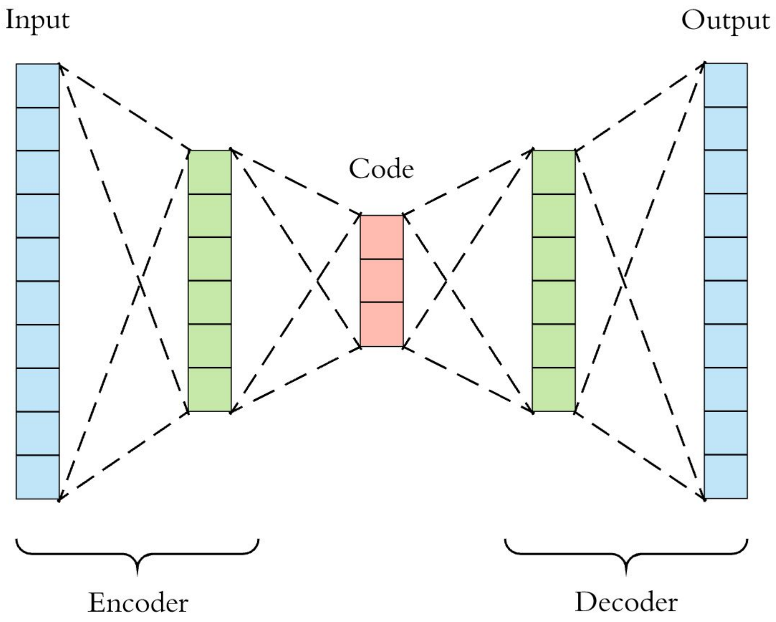


Intro Deep Learning - Part 3

Aut

-
-
-
-

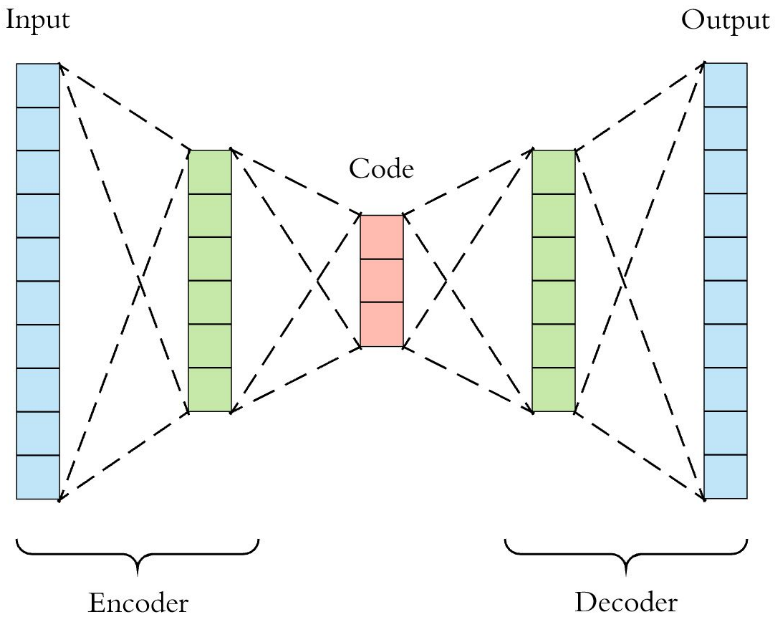
```python
class Autoencoder(nn.Module):
    def __init__(self):
        super(Autoencoder, self).__init__(input_size)
        self.encoder = nn.Sequential(
            nn.Linear(input_size, 128),
            nn.ReLU(),
            nn.Linear(128, 64),
            nn.ReLU(),
        )
        self.decoder = nn.Sequential(
            nn.Linear(64, 128),
            nn.ReLU(),
            nn.Linear(128, input_size)
        )

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x
```



Intro Deep Learning - Part 3

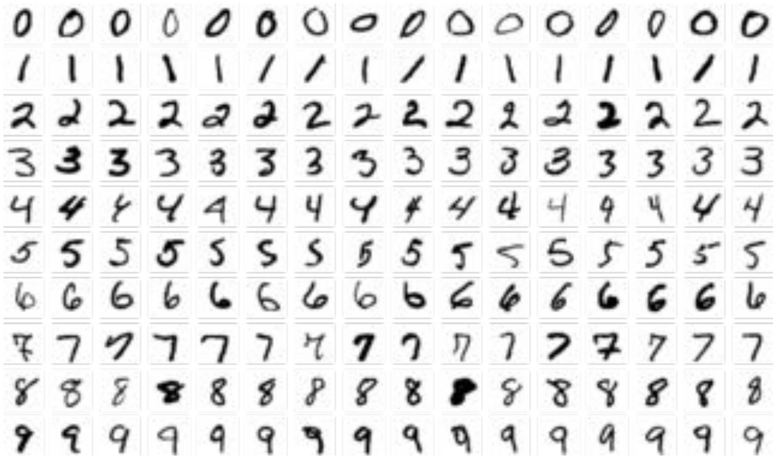
Auto-encoders

- Learns features
- Data compression
- Reduces dimensions
- Unsupervised Learning approach



MNIST

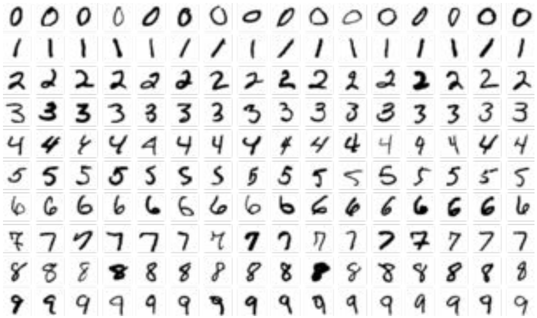
- Dataset of grayscale images of hand-written numbers 0-9
- Used to benchmark image classification architectures

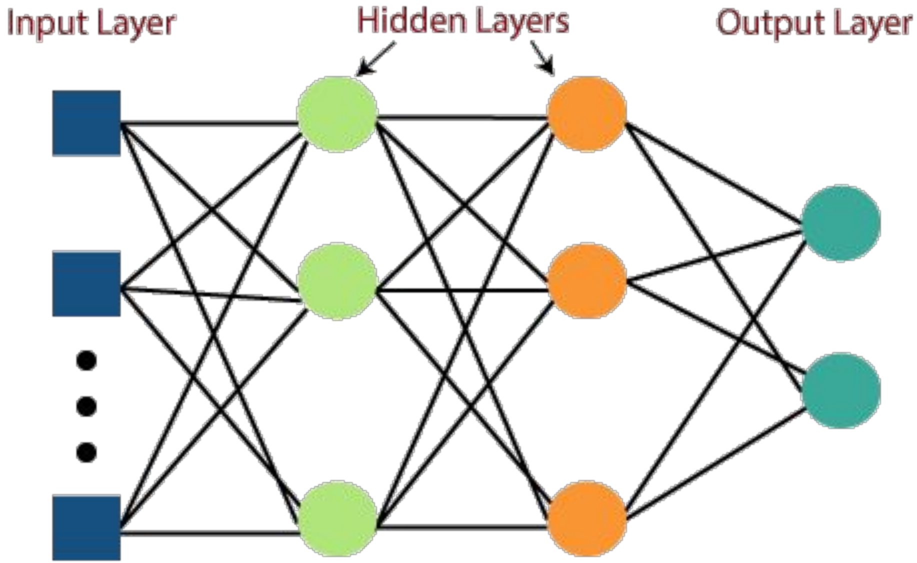


Intro Deep Learning - Part 3

MNIST

- Dataset of grayscale images of hand-written numbers 0-9
- Used to benchmark image classification architectures

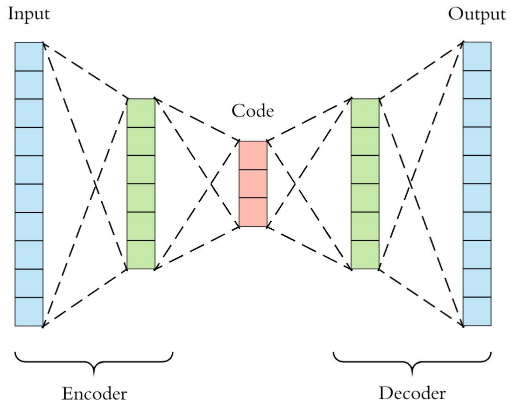


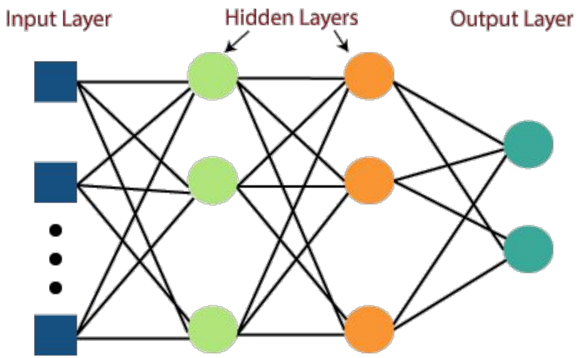


Intro Deep Learning - Part 3

MNIST

- Dataset of grayscale images of hand-written numbers 0-9
- Used to benchmark image classification architectures

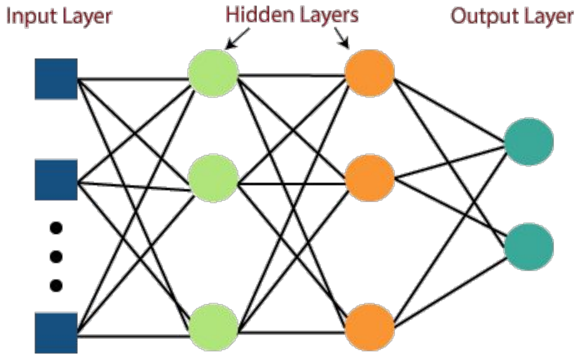




Intro Deep Learning - Part 3

What about text?

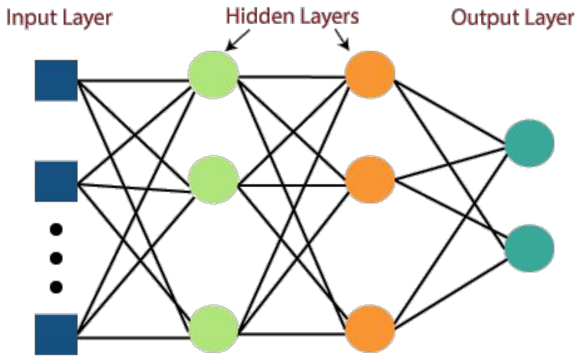
- How do we feed text into a deep learning architecture ?



Intro Deep Learning - Part 3

What about text?

- How do we feed text into a deep learning architecture ?
- Embedding layer
  - “A simple lookup table that stores embeddings of a fixed dictionary and size.”
  - Converts a token into a embedding
  - Embedding is the input of the next layer



Intro Deep Learning - Part 3

What about text?

- How do we feed text into a deep learning architecture ?
- Embedding layer
  - “A simple lookup table that stores embeddings of a fixed dictionary and size.”
  - Converts a token into a embedding
  - Embedding is the input of the next layer

```
>>> # an Embedding module containing 10 tensors of size 3
>>> embedding = nn.Embedding(10, 3)
>>> # a batch of 2 samples of 4 indices each
>>> input = torch.LongTensor([[1, 2, 4, 5], [4, 3, 2,
9]])embedding(input)
tensor([[[-0.0251, -1.6902,  0.7172],
         [-0.6431,  0.0748,  0.6969],
         [ 1.4970,  1.3448, -0.9685],
         [-0.3677, -2.7265, -0.1685]],

        [[ 1.4970,  1.3448, -0.9685],
         [ 0.4362, -0.4004,  0.9400],
         [-0.6431,  0.0748,  0.6969],
         [ 0.9124, -2.3616,  1.1151]]])
```

Intro Deep Learning - Part 3

Wh

```python
import torch
import torch.nn as nn

class TextClassifier(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, num_classes):
        super(TextClassifier, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.fc1 = nn.Linear(embedding_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, num_classes)

    def forward(self, x):
        embedded = self.embedding(x)
        # Flatten the 3D tensor into a 2D tensor
        embedded = embedded.view(embedded.size(0), -1)
        x = torch.relu(self.fc1(embedded))
        logits = self.fc2(x)
        return logits
```

```python
 an Embedding module containing 10 tensors of size 3
mbedding = nn.Embedding(10, 3)
 a batch of 2 samples of 4 indices each
nput = torch.LongTensor([[1, 2, 4, 5], [4, 3, 2,
mbedding(input)
r([[[-0.0251, -1.6902,  0.7172],
    [-0.6431,  0.0748,  0.6969],
    [ 1.4970,  1.3448, -0.9685],
    [-0.3677, -2.7265, -0.1685]],

   [[ 1.4970,  1.3448, -0.9685],
    [ 0.4362, -0.4004,  0.9400],
    [-0.6431,  0.0748,  0.6969],
    [ 0.9124, -2.3616,  1.1151]]])
```

Intro Deep Learning - Part 3

Regularization

- Goal of preventing overfit, improving generalization and robustness
- L1 and L2
- Dropout
- Batch Normalization
- Early Stopping

Intro Deep Learning - Part 3

Regularization

- Goal of preventing overfit, improving generalization and robustness
- L1
- L2
- Dropout
- Batch Normalization
- Early Stopping

L1 adds a penalty to the loss function proportional to the absolute values of the models parameters

$$Loss = Error(y, \hat{y}) + \lambda \sum_{i=1}^{N} |w_i|$$

Hyperparameter

Intro Deep Learning - Part 3

Regularization

- Goal of preventing overfit, improving generalization and robustness
- L1
- L2
- Dropout
- Batch Normalization
- Early Stopping

```
l1_reg = torch.tensor(0., requires_grad=True)
for param in model.parameters():
    l1_reg = l1_reg + torch.norm(param, 1)

total_loss = loss + l1_lambda * l1_reg
total_loss.backward()
```

Regularization

- Goal of preventing overfit, improving generalization and robustness
- L1
- L2
- Dropout
- Batch Normalization
- Early Stopping

L2 regularization adds a penalty term to the loss function that is proportional to the square of the model's parameters.

$$Loss = Error(y, \hat{y}) + \lambda \sum_{i=1}^{N} w_i^2$$

Hyperparameter

https://towardsdatascience.com/l1-and-l2-regularization-methods-ce25e7fc831c

Intro Deep Learning - Part 3

Regularization

- Goal of preventing overfit, improving generalization and robustness
- L1
- L2
- Dropout
- Batch Normalization
- Early Stopping

L1 regularization encourages **sparsity** in model parameters, making it useful for **feature selection**.

L2 regularization **discourages large parameter values**, helping to prevent overfitting and improve the generalization of models.

They can be combined, which is commonly referred to as Elastic Net regularization,

https://towardsdatascience.com/l1-and-l2-regularization-methods-ce25e7fc831c

Intro Deep Learning - Part 3

Regularization

- Goal of preventing overfit, improving generalization and robustness
- L1
- L2
- Dropout
- Batch Normalization
- Early Stopping

```python
import torch.nn as nn

class MyModel(nn.Module):
    def __init__(self):
        super(MyModel, self).__init__()
        self.fc1 = nn.Linear(128, 64)
        self.dropout = nn.Dropout(0.5)  # Dropout layer with a dropout rate of 0.5
        self.fc2 = nn.Linear(64, 10)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.dropout(x)  # Apply dropout to the output of fc1
        x = self.fc2(x)
        return x
```

Intro Deep Learning - Part 3

Regularization

- Goal of preventing overfit, improving generalization and robustness
- L1
- L2
- Dropout
- Batch Normalization
- Early Stopping

```python
import torch.nn as nn

class MyModel(nn.Module):
    def __init__(self):
        super(MyModel, self).__init()
        self.fc1 = nn.Linear(128, 64)
        self.bn1 = nn.BatchNorm1d(64)  # Batch Normalization layer
        self.fc2 = nn.Linear(64, 10)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.bn1(x)  # Apply BatchNorm to the output of fc1
        x = self.fc2(x)
        return x
```

Improves training stability
Reduces internal covariate shift

Intro Deep Learning - Part 3

Regularization

- Goal of preventing overfit, improving generalization and robustness
- L1
- L2
- Dropout
- Batch Normalization
- Early Stopping

```python
best_validation_metric = float('inf')  # Initialize with a large value
patience = 5  # Number of consecutive evaluations with no improvement allowed
no_improvement_count = 0

for epoch in range(max_epochs):
    # Training steps
    # ...

    # Validation step
    validation_metric = compute_validation_metric(model, validation_data)

    if validation_metric < best_validation_metric:
        best_validation_metric = validation_metric
        no_improvement_count = 0
    else:
        no_improvement_count += 1

    if no_improvement_count >= patience:
        print(f"Early stopping at epoch {epoch}.")
        break
```

Intro Deep Learning - Part 3

Regularization

- Goal of preventing overfit, improving generalization and robustness
- L1
- L2
- Dropout
- Batch Normalization
- Early Stopping

```python
best_validation_metric = float('inf')  # Initialize with a large value
patience = 5  # Number of consecutive evaluations with no improvement allowed
no_improvement_count = 0

for epoch in range(max_epochs):
    # Training steps
    # ...

    # Validation step
    validation_metric = compute_validation_metric(model, validation_data)

    if validation_metric < best_validation_metric:
        best_validation_metric = validation_metric
        no_improvement_count = 0
    else:
        no_improvement_count += 1

    if no_improvement_count >= patience:
        print(f"Early stopping at epoch {epoch}.")
        break
```

Early Stopping Criteria

Wrap up

Architectures

- MLP
- CNN
- RNN/LSTM
- GNNs
- Transformers
- …
- And a bunch of variations of each

Frameworks

- Tensorflow/Keras
- FastAI
- LightningAI

Hardware

- CPU
- GPU
- TPU