ECE 364: Programming Methods for Machine Learning, Spring 2025

Midterm 2 – May 01, 2025

- **You will have 75 minutes (1.25 hours) to solve all the problems. Most have multiple parts.** Don't spend too much time on questions you don't understand and focus on answering as much as you can!
- ***BUDGET YOUR TIME WISELY***. I highly recommend working on the questions you know first and the questions you need to think about second.
- *No* resources are allowed for use during the exam except a cheatsheet and scratch paper on the back of the exam. **Do not tear out the cheatsheet or the scratch paper!** It messes with the auto-scanner.
- You should write your answers *completely* in the space given for the question. We will not grade part

  s of any answer written outside of the designated space.
- Please *use a dark-colored pen* unless you are *absolutely* sure your pencil writing is forceful enough to be legible when scanned. We reserve the right to take off points if we have difficulty reading the uploaded document.
- **Don't cheat.** C'mon, be cool, be honest.
- **Good luck!**

Name: ____________________________

NetID: ____________________________

Date: ____________________________

1. **Fact or Cap?** (30 points)

For each question, circle whether the statement is true or false.

(a) True **FALSE** The first principal component captures the lowest variance of the dataset.

(b) True **FALSE** In each iteration of K-Means, centroids are updated by moving them toward the point with the smallest distance in their cluster.

(c) True **FALSE** In a convolutional layer, each weight (or parameter) is used exactly once while computing the output of the layer.

(d) **TRUE** False A strong discriminator too early in GAN training would make it difficult for the generator to improve.

(e) **TRUE** False The sigmoid activation function can cause vanishing gradients in deep networks.

(f) True **FALSE** The derivative of ReLU is always 1.

(g) **TRUE** False Standard RNNs are designed to handle sequential data by maintaining a hidden state across time steps.

(h) True **FALSE** LSTMs suffer more from the vanishing gradient problem than standard RNNs do.

(i) **TRUE** False Self-attention compares every token to every other token in the sequence

(j) True **FALSE** A convolutional layer applies different kernels (filters) to different regions of the input.

(k) **TRUE** False Masked language modeling is used to pre-train BERT by randomly masking input tokens.

(l) **TRUE** False In decoder self-attention, a causal mask prevents tokens from attending to future positions.

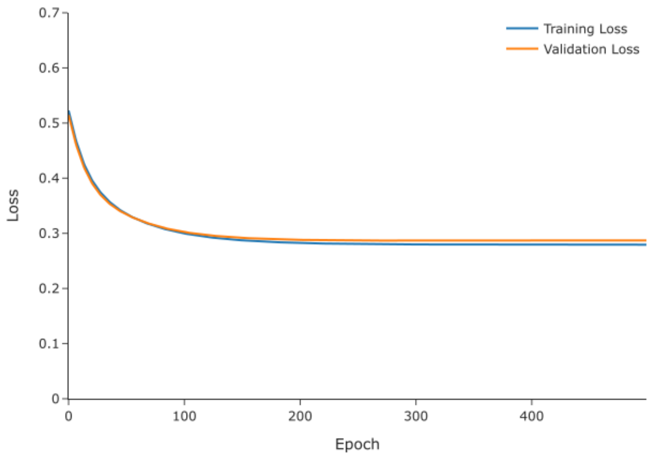
(m) **TRUE** False Decoder-only transformer models have no cross-attention mechanism.

(n) True **FALSE** Single shot multibox detection uses the same default-box scale for every feature map level.

(o) **TRUE** False In deep NNs, having multiple linear layers back to back without an activation function in between is mathematically no different than having one linear layer.

2. **It Trained All Night and Still Flopped** (10 points)

In your machine learning course, you have been assigned to build a classification model to detect whether emails are spam or not. After dedicating several days to the project, you have designed a promising model. But, during training, you observe the following behavior in the training and validation losses:



(a) You know your classmates have been able to achieve lower losses than what you have plateaued at. Which of the following is a possible issue with the training? Circle your answer.

Underfitting Overfitting No issue

Solution:
Underfitting

(b) Justify your answer. If you selected "Underfitting" or "Overfitting", suggest a possible remedy to mitigate the issue. Use no more than four sentences in total for your answer.

Solution:
Both training and validation losses converge to a high value. This suggests underfitting. The model is not complex enough for the task. A possible remedy would be to increase the number of parameters in the model. Replacing linear activation functions with non-linear ones would also help.

3. **Life's a Pool, Dive In** (10 points)

Consider the 5×5 input

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 4 & 3 & 2 & 1 & 0 \\ 1 & 2 & 3 & 4 & 5 \\ 5 & 4 & 3 & 2 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{pmatrix}.$$

Compute the output of `MaxPool2d(3, stride=2)`.

Solution:

$$\begin{pmatrix} 4 & 5 \\ 5 & 5 \end{pmatrix}.$$

4. **Who Wants to Be a Parameterionaire?** (10 points)

Consider the following network:

```
model = nn.Sequential(
    nn.Linear(8, 16, bias=True),
    nn.ReLU(),
    nn.Linear(16, 4, bias=True)
)
```

(a) Suppose the input is of shape $(1, 8)$. What are the shapes after the first Linear, after ReLU, and after the second Linear?

Solution:
After the first Linear: $(1, 16)$
ReLU: $(1, 16)$
the second Linear: $(1, 4)$

(b) Determine the total learnable parameters in the model.

Solution:
First linear layer: $8 \times 16 + 16 = 144$
Second linear layer: $16 \times 4 + 4 = 68$
Total: $144 + 68 = 212$ parameters

5. **Layer by Layer: A Neural Network Love Story** (10 points)

In this problem, you will be hand designing a simple neural network to model a specific function. Assume $x \in \mathbb{R}$ and provide appropriate weights $w_0, w_1, b_0 \in \mathbb{R}^2$. In other words, the neural network has one input neuron, 2 hidden neurons, and one output neuron.

Find $w_0, w_1, b_0 \in \mathbb{R}^2$ such that $f(x) = w_1^T \sigma(w_0 x + b_0) = mx + b, \forall x \in \mathbb{R}$ where $\sigma = ReLU$ (note: $w_0 x$ here is a vector-scalar product: $\mathbb{R}^2 \times \mathbb{R} \to \mathbb{R}^2$, e.g. $[0,1]^T x = [0,x]^T$). Show why your answer is correct.

Solution:

Note multiple solutions may exist. In order to achieve this function, we can set $w_0 = [m, -m]^T, w_1 = [1,-1]^T, b_0 = [b, -b]^T$. We can see that this works because applying our answer to our function, we get

$$f(x) = w_1^T \sigma(w_0 x + b_0) = [1,-1]^{T^T} \sigma([m,-m]^T x + [b,-b]^T) = [1,-1]\sigma([mx+b, -mx-b]^T)$$

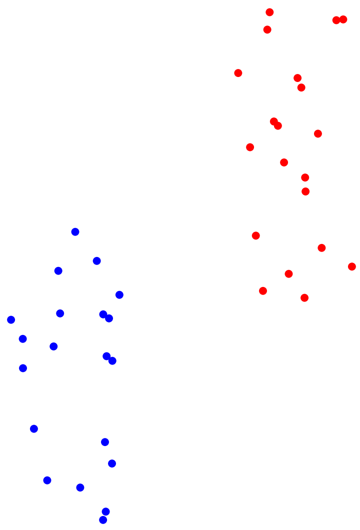
$$= [1,-1]\sigma([mx+b, -(mx+b)]^T) = \begin{cases} [1,-1][mx+b,0]^T = mx+b & , mx+b > 0 \\ [1,-1][0,0]^T = 0 & , mx+b = 0 \\ [1,-1][0,-mx-b]^T = mx+b & , mx+b < 0 \end{cases}$$

$$\Rightarrow f(x) = mx + b, \forall x \in \mathbb{R}$$

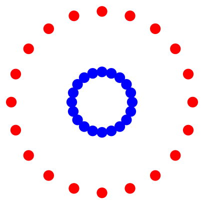
(ReLU: $\sigma(x) = max(0,x)$).

6. **Two Clusters Walk Into a Bar. . .** (15 points)

(a) For each figure below, indicate whether it is possible for K-Means, GMM, neither, or both algorithms to produce the clustering assignments shown (as indicated by the two colors which are there for visualization purposes only). In 1–2 sentences, briefly explain your reasoning.



(a)



(b)

Solution:

(a) Both

(b) Neither

(b) Which algorithm is more sensitive to outliers: K-Means or GMM? Explain.

Solution:
K-Means is more sensitive because it uses means, which shift due to outliers. GMM can be more robust if the outlier is far enough to get very low probability under all components.

(c) Which of the following is *not* minimized (or maximized) by K-Means or GMM?

A. $\sum_{i=1}^{n} \min_k \|x_i - \mu_k\|^2$

B. $-\sum_{i=1}^{n} \log\Big(\sum_{k=1}^{K} \pi_k \, \mathcal{N}(x_i \mid \mu_k, \Sigma_k)\Big)$

C. $\sum_{i=1}^{n} \min_k \|x_i - \mu_k\|$

D. $\sum_{i=1}^{n} \sum_{k=1}^{K} \gamma_{ik} \, \|x_i - \mu_k\|^2$

Solution:
(c). K-Means uses *squared* distances, and GMM optimizes a likelihood, neither optimizes the sum of *unsquared* distances.

7. **Attention is All You Need!** (15 points)

The Transformer architecture integrates attention mechanisms with multi-layer perceptrons (MLPs). The attention mechanism takes three inputs – queries (Q), keys (K), and values (V). Each has a shape of $B \times N \times d_{\text{model}}$. Here, B, N, and d_{model} represent the batch size, sequence length, and model dimension (or hidden size), respectively. In this question, you will implement a simple attention mechanism without using multiple heads. The attention outputs are computed as follows.

Step 1. We first apply learned projection matrices as

$$Q_w = QW^Q + b^Q, K_w = KW^K + b^K, V_w = VW^V + b^V$$

where $W^Q, W^K, W^V \in \mathbb{R}^{d_{\text{model}} \times d_{\text{model}}}$ and $b^Q, b^K, b^V \in \mathbb{R}^{d_{\text{model}}}$.

Step 2. The attention outputs are computed as

$$A = \text{softmax}\left(\frac{Q_w K_w^T}{\sqrt{d_{\text{model}}}}\right) V_w$$

Step 3. Apply another linear projection on A

$$\text{Attention}(Q, K, W) = AW^O + b^O$$

where $W^O \in \mathbb{R}^{d_{\text{model}} \times d_{\text{model}}}$ and $b \in \mathbb{R}^{d_{\text{model}}}$.

Complete the code below to implement attention. Use the provided definitions as a guide. For this question, you can ignore additional details such as attention masks, multi-head attention, dropouts, and optimized implementations.

```python
# Even though you should not need to import anything else, feel
# free to do so.
import torch
import torch.nn as nn
from torch.nn.functional import softmax





class Attention(nn.Module):
    def __init__(self, d_model: int) -> None:
        self.d_model = d_model
        # Space to add more class variables as required













    def forward(self, Q: torch.Tensor, K: torch.Tensor, V: torch.Tensor) ->
    torch.Tensor:
        """
        Inputs
        ------
        Q: B x N x d_model
        K: B x N x d_model
        V: B x N x d_model
        """
        # Complete this













































        # Function end
```

Solution:

```python
# Even though you should not need to import anything else, feel
# free to do so.
import torch
import torch.nn as nn
from torch.nn.functional import softmax

class Attention(nn.Module):
    def __init__(self, d_model: int) -> None:
        self.d_model = d_model

        # Space to add more class variables as required
        self.wq = nn.Linear(self.d_model, self.d_model, bias=True)
        self.wk = nn.Linear(self.d_model, self.d_model, bias=True)
        self.wv = nn.Linear(self.d_model, self.d_model, bias=True)
        self.wo = nn.Linear(self.d_model, self.d_model, bias=True)
        self.scaling = self.d_model**-0.5

    def forward(self, Q: torch.Tensor, K: torch.Tensor, V: torch.Tensor) ->
    torch.Tensor:
        """
        Inputs
        ------
        Q: B x N x d_model
        K: B x N x d_model
        V: B x N x d_model
        """
        # Complete this
        B, N, d_model = Q.size()
        query_states = self.wq(Q) * self.scaling # B x N x d_model
        key_states = self.wk(K) # B x N x d_model
        val_states = self.wv(V) # B x N x d_model

        attn_weights = query_states @ key_states.transpose(1, 2) # B x N x N
        attn_weights = softmax(attn_weights, dim=-1) # B x N X N

        attn_outputs = attn_weights @ value_states # B x N x d_model
        attn_outputs = self.wo(attn_outputs) # B x N x d_model

        return attn_outputs
        # Function end
```

This page is for additional scratch work!

This page is for additional scratch work!

Deep Learning Cheatsheet

torch.nn Functions

- **Linear function:** $y = WX + b$ where W and X are vectors of size N (number of dimensions to the input).

```
torch.nn.Linear(in_features, out_features, bias=True, device=None, dtype=None)
```

- **Sigmoid function:** $\frac{1}{1+e^{-z}}$ where z is the logit(s).

```
torch.nn.functional.sigmoid(input)
```

- **Softmax function:** $p(Y = t|x) = \frac{\exp(w_t^T x)}{\sum_{y\in\{0,\ldots,C-1\}} \exp(w_y^T x)}$

```
torch.nn.functional.softmax(input, dim=None, _stacklevel=3, dtype=None)
```

torch.nn Layers

- **Linear layer:** $y = WX + b$ where W and X are vectors of size N (number of dimensions to the input).

```
torch.nn.Linear(in_features, out_features, bias=True, device=None, dtype=None)
```

- **Convolutional layer:** In the simplest case, assuming a input size of (N, C_{in}, H, W), the output is sized$(N, C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$ where N is a batch size, C denotes a number of channels, H is a height of input planes in pixels, and W is width in pixels.

```
torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1,
    bias=True, padding_mode='zeros', device=None, dtype=None)
```

  (1d/2d/3d variations available as well)

- **Pooling layer:** Applies a #D (1d/2d/3d variations available) pooling over an input signal composed of several input planes. There are two flavors

```
torch.nn.MaxPool2d(kernel_size, stride=None, padding=0, dilation=1, return_indices=False, ceil_mode
    =False)
```

```
torch.nn.AvgPool2d(kernel_size, stride=None, padding=0, ceil_mode=False, count_include_pad=True,
    divisor_override=None)
```

- **BatchNorm layer:** normalizes the data over a batch using the formula $y = \frac{x-\mathbb{E}[x]}{\sqrt{\text{Var}[x]+\epsilon}} * \gamma + \beta$ where γ and β are trainable parameters:

```
torch.nn.BatchNorm2d(num_features, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True,
    device=None, dtype=None)
```

Principal Component Analysis

Steps to calculate principal components:

- Subtract means and calculate covariance matrix: $\mathbf{\Sigma} = \frac{1}{N}\sum_{n=1}^{N} \mathbf{x}^{(n)} \left(\mathbf{x}^{(n)}\right)^T = \frac{1}{N} X^T X$
- Find the D eigenvectors with the largest eigenvalues:

```
eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)
# Use torch.pca_lowrank to compute the top 2 principal components
U, S, V = torch.pca_lowrank(X, q=2)
# Project the data onto the principal components
X_pca = X @ V
```

K-means clustering

Steps to calculate k-means clusters. First, we got to choose how many (K) clusters we want to break the data up into. Randomly assign data points into clusters.

- Update centroid of clusters based on current data point assignment.
- Update datapoint assignment based off cluster centroids.

Loss function: $J\left(C_1, \ldots, C_k, \mu_1, \ldots, \mu_k\right) = \Sigma_{k=1}^{K} \Sigma_{i\in C_k} ||x^{(i)} - \mu_k||^2$

Gaussian mixture models

Model the data set as a combination of Gaussian curves: $p(x) = \sum_{k=1}^{K} \pi_k \, \mathcal{N}(x \mid \mu_k, \Sigma_k)$ where $\mathcal{N}(x \mid \mu_k, \Sigma_k)$: Gaussian density for the k-th component, $\mathcal{N}(x \mid \mu_k, \Sigma_k)$: Gaussian density for the k-th component, and K: total number of components.

- **Expectation step:** compute the "responsibilities" or the posterior probabilities that a data point $x^{(i)}$ belongs to each Gaussian component k: $\gamma(z_k^{(i)}) = \frac{\pi_k \, \mathcal{N}(x^{(i)} | \mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j \, \mathcal{N}(x^{(i)} | \mu_j, \Sigma_j)}$
- **Maximization step:** parameters of the GMM (i.e., the mixing coefficients, means, and covariances) are updated to maximize the expected complete-data log-likelihood ($\sum_{i=1}^{N} \log \left(\sum_{k=1}^{K} \pi_k \, \mathcal{N}(x^{(i)} \mid \mu_k, \Sigma_k) \right)$) computed during the E-Step.
  - Update mixing coefficient: $\pi_k = \frac{1}{N} \sum_{i=1}^{N} \gamma(z_k^{(i)})$
  - Update means: $\mu_k = \frac{\sum_{i=1}^{N} \gamma(z_k^{(i)}) \, x^{(i)}}{\sum_{i=1}^{N} \gamma(z_k^{(i)})}$
  - Update covariance matrices: $\Sigma_k = \frac{\sum_{i=1}^{N} \gamma(z_k^{(i)}) \, (x^{(i)} - \mu_k)(x^{(i)} - \mu_k)^\top}{\sum_{i=1}^{N} \gamma(z_k^{(i)})}$

Generative adversarial networks

Assume a generator ($G_\theta(z)$) and discriminator ($D_w(x) = p(y = 1|x)$). The loss function for the discriminator is: $\mathcal{J}_D = -\Sigma_x \log D_w(x) - \Sigma_z \log(1 - D_w(G_\theta(z)))$.
For the generator, we have the formulation: $\mathcal{J}_G = -\mathcal{J}_D = const + \Sigma_z \log(1 - D_w(G_\theta(z)))$, but the question is how to optimize:

- min-max formulation: $\max_\theta \min_w -\Sigma_x \log D_w(x) - \Sigma_z \log(1 - D_w(G_\theta(z)))$
- non-saturating formulation: $\min_\theta -\Sigma_z \log(D_w(G_\theta(z)))$

Image processing

- For bounding box problems, we optimize intersection over union.
- Let $x_{ij}^p \in \{0, 1\}$ be an indicator of matching default box i to ground-truth box j from class p, c be the class of the bounding box, l be the predicted bounding box, g be the ground-truth bounding box, and d be the matched default box. The loss function $\mathcal{L}$ is given as $\mathcal{L}(x, c, l, g) = \frac{1}{N} \left(\mathcal{L}_{\text{cls}}(x, c) + \alpha \mathcal{L}_{\text{loc}}(x, l, g) \right)$, where N is the number of matched default boxes for the given image

Accuracy measures:

- $\text{Precision} = \frac{TP}{TP+FP} \in [0, 1]$, $\text{Recall} = \frac{TP}{TP+FN} \in [0, 1]$
- Average precision (AP) is area under precision recall curve.
- mAP score is AP at multiple IoU thresholds.

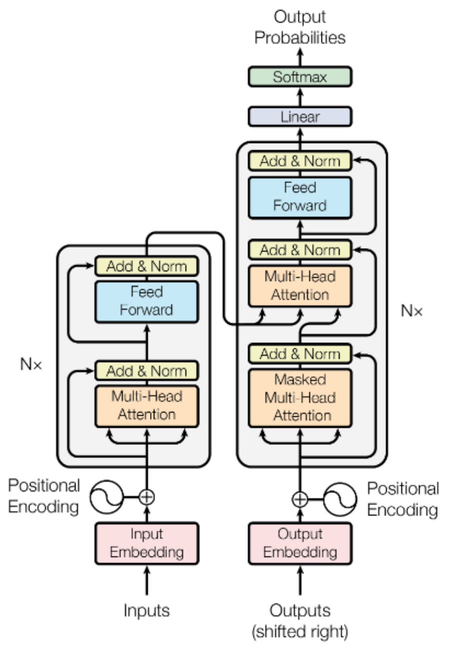


Figure 1: The Transformer - model architecture.

Transformer model

Position encoding:

- $\text{PE}^0_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}} \right)$
- $\text{PE}^1_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}} \right)$

where pos is the position in the sequence, i is the embedding dimension index is the d_{model} = total embedding size (e.g., 512) **Attention types:**

- $\text{SelfAttention}(Q_{\text{encoder}}, K_{\text{encoder}}, V_{\text{encoder}}) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}} \right) V$
- $\text{CrossAttention}(Q_{\text{decoder}}, K_{\text{encoder}}, V_{\text{encoder}}) == \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}} \right) V$
- $\text{MaskedAttention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top + M}{\sqrt{d_k}} \right) V$ where M is a mask of $-\infty$ values.

Attention heads are calculated by: $\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$ and $\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h) W^O$

Sample Code - Logistic classification

Here is a sample, two-dimensional logistic classifier code:

```python
import numpy as np
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
from torch.utils.data import SubsetRandomSampler

class LogisticRegression(nn.Module):
    def __init__(self, N):
        super().__init__()
        self.w = nn.Parameter(torch.ones(N))
        self.b = nn.Parameter(torch.zeros(1))

    def forward(self, x):
        return 1/(1+torch.exp(-(self.w@x+self.b)))

class TwoClassDataset(Dataset):
    # don't forget the self identifier!
    def __init__(self, N, sigma):
        self.N = N # number of data points per class
        self.sigma = sigma # standard deviation of each class cluster
        self.plus_class = self.sigma*torch.randn(N, 2) + torch.tensor([-1, 1])
        self.negative_class = self.sigma*torch.randn(N, 2) + torch.tensor([1, -1])
        self.data = torch.cat((self.plus_class, self.negative_class), dim=0)
        self.labels = torch.cat((torch.ones(self.N), torch.zeros(self.N)))

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

    def __getitem__(self, idx):
        x = self.data[idx]
        y = self.labels[idx]
        return x, y # return input and output pair

N = 100
sigma = 1.5
dataset = TwoClassDataset(N, sigma)
plus_data = dataset.plus_class
negative_data = dataset.negative_class

# create indices for each split of dataset
N_train = 60
N_val = 20
N_test = 20
indices = np.arange(len(dataset))
np.random.shuffle(indices)
train_indices = indices[:N_train]
val_indices = indices[N_train:N_train+N_val]
test_indices = indices[N_train+N_val:]

# create dataloader for each split
batch_size = 8
train_loader = DataLoader(dataset, batch_size=batch_size, sampler=SubsetRandomSampler(train_indices)
    )
val_loader = DataLoader(dataset, batch_size=batch_size, sampler=SubsetRandomSampler(val_indices))
test_loader = DataLoader(dataset, batch_size=batch_size, sampler=SubsetRandomSampler(test_indices))

criterion = nn.BCELoss(reduction='mean') # binary cross-entropy loss, use mean loss
logreg_model = LogisticRegression(2) # initialize model
optimizer = torch.optim.SGD(logreg_model.parameters()) # initialize optimizer

n_epoch = 200 # number of passes through the training dataset
loss_values, train_accuracies, val_accuracies = [], [], []
for n in range(n_epoch):
    epoch_loss, epoch_acc = 0, 0
    for x_batch, y_batch in train_loader:
        optimizer.zero_grad()
        predictions = logreg_model(x_batch.unsqueeze(-1)).squeeze(-1)
        loss = criterion(predictions, y_batch)
        loss.backward()
        optimizer.step()
```

Sample Code - Linear network

Here is a sample, linear torch.nn network model:

```python
class ThreeLayerMLP(nn.Module):
    def __init__(self, input_dim, h1, h2, output_dim, activation_fn):
        super().__init__()
        self.fc1 = nn.Linear(input_dim, h1) # first fully-connected layer
        self.fc2 = nn.Linear(h1, h2) # second fully-connected layer
        self.fc3 = nn.Linear(h2, output_dim) # output layer
        self.activation = activation_fn

    def forward(self, x):
        x = self.activation(self.fc1(x)) # first layer
        x = self.activation(self.fc2(x)) # second layer
        z = self.fc3(x) # output layer
        return z
```

Sample Code - convolutional neural network

Here is a sample, linear torch.nn network model:

```python
class MyCNNModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1, bias=False)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1, bias=False)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1, bias=False)
        self.pooling = nn.MaxPool2d(stride=2, kernel_size=2)
        self.gap = nn.AdaptiveAvgPool2d((1, 1))
        self.activation = nn.ReLU()
        self.fc = nn.Linear(128, 10)

    def forward(self, x):
        B = x.size(0)
        x = self.activation(self.conv1(x))
        x = self.pooling(x)
        x = self.activation(self.conv2(x))
        x = self.pooling(x)
        x = self.activation(self.conv3(x))
        x = self.gap(x).view(B, -1)
        z = self.fc(x)
        return z
```