ECE 364: Programming Methods for Machine Learning, Spring 2026

Midterm 2 – April 30, 2026

- **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.
- No electronic devices (phones, watches, smart glasses, etc.) should be out. Put it all away.
- **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** In a 2D convolutional layer, setting padding $P = (K-1)/2$ with stride 1 always guarantees that the output feature map has the same height and width as the input, regardless of the kernel size K.

(b) True **FALSE** A multi-layer perceptron trained on FashionMNIST will achieve noticeably worse accuracy if you first apply a fixed random permutation to every image's pixels (the same permutation for all images) before training.

(c) True **FALSE** When test images are shifted 2 pixels to the right, MLPs and CNNs suffer an equal drop in accuracy, because neither architecture was explicitly trained on shifted data.

(d) True **FALSE** In PCA, the principal component is the eigenvector of the data covariance matrix corresponding to the *smallest* eigenvalue.

(e) True **FALSE** K-Means and the Gaussian Mixture Model trained with EM are both hard clustering methods: each data point is ultimately assigned to exactly one cluster in both algorithms.

(f) True **FALSE** During evaluation (`model.eval()`), both Dropout and Batch Normalization are effectively turned off, meaning neither layer has any effect on the forward pass.

(g) **TRUE** False In the GMM E-step, the responsibilities γ_{nk} for a single data point n sum to 1 across all components k.

(h) **TRUE** False In a Generative Adversarial Network, the generator tries to create realistic data samples while the discriminator tries to distinguish between real and generated data.

(i) True **FALSE** During GAN training, the generator and discriminator are trained completely independently and do not affect each other's learning.

(j) **TRUE** False In deep learning training, batch normalization can address the problem caused by internal covariate shift.

(k) **TRUE** False In single-stage object detection models like SSD, each location in a feature map is associated with a predefined anchor (default) associated with boxes of different scales and aspect ratios to detect objects of various sizes.

(l) **TRUE** False Tokenization can segment language by letters, words or something in between.

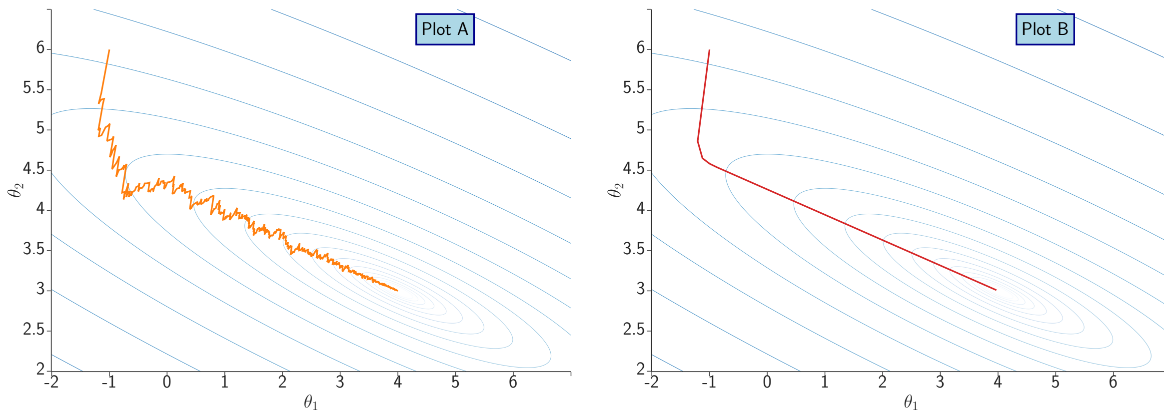
(m) **TRUE** False In the attention mechanism, queries, keys, and values are all derived from the same input in self-attention.

(n) True **FALSE** In decoder of Transformer, queries, and keys used in cross-attention are derived from encoder.

(o) **TRUE** False Multi-head attention uses multiple attention heads to allow the model to capture different types of relationships in parallel.

2. **Race to the Bottom** (10 points)

Consider a dataset $\{x_i, y_i\}_{i=1}^{N}$ that is generated using some process $y = f(x)$. Here, $x_i, y_i \in \mathbb{R}$. We do not know the true f and only have access to N points generated through this process. We have approximated the process with a linear model, $\hat{y} = \theta_0 + \theta_1 x$, and used two optimization algorithms to learn the parameters: Gradient Descent (GD) and Stochastic Gradient Descent (SGD). The plots below show the learned values of θ_0 and θ_1 over time. Both runs start from the point indicated by "Start" and end at the point indicated by "End".



(a) Which of the above plots is most likely for SGD? Circle your answer.

Plot A Plot B Cannot be determined

Solution:
Plot A

(b) Justify your answer. Use no more than four sentences in total for your answer.

Solution:
SGD uses a single example to compute the gradient at each step. As it does not represent the true gradient of the dataset at a given step, the parameter updates will be noisy, causing the optimization path to fluctuate rather than moving directly toward a minimum.

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

Consider the 5×5 input

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

and the 3×3 filter

$$\begin{pmatrix} 1 & 0 & -1 \\ 0 & 1 & 0 \\ -1 & 0 & 1 \end{pmatrix}$$

Compute the output of the convolution with stride = 2 and no padding.

Solution:

$$\begin{pmatrix} 0 & 2 \\ 1 & -1 \end{pmatrix}.$$

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

Consider the following network:

```
model = nn.Sequential(
    nn.Linear(10, 12, bias=True),
    nn.Tanh(),
    nn.Linear(12, 6, bias=False) ,
    nn.ReLU(),
    nn.Linear(6, 3, bias=True)
)
```

(a) Suppose the input is of shape $(1, 10)$. What are the shapes after each Linear (First Linear, Tanh, Second Linear, ReLU, Third Linear)?

Solution:
After the first Linear: $(1, 12)$
Tanh: $(1, 12)$
the second Linear: $(1, 6)$
ReLU: $(1, 6)$
Third Linear: $(1, 3)$

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

Solution:
First linear layer: $10 \times 12 + 12 = 132$
Second linear layer: $12 \times 6 = 72$
Third linear layer: $6 \times 3 + 3 = 21$
Total: $132 + 72 + 21 = 225$ parameters

5. **Layer by Layer: The Kink Awakens** (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 $f(x) = w_1^T \sigma(w_0 x + b_0) = |x|, \forall x \in \mathbb{R}$ 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:

In order to achieve this function, we can set $w_0 = [1,-1]^T, w_1 = [1,1]^T, b_0 = [0,0]^T$. We can see that this works because applying our answer to our function, we get

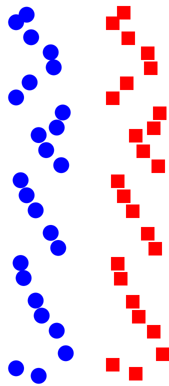
$$f(x) = w_1^T \sigma(w_0 x + b_0) = [1,1]^{T^T} \sigma([1,-1]^T x + [0,0]^T) = [1,1]\sigma([x,-x]^T)$$

$$= \begin{cases} [1,1][x,0]^T = x = |x| & ,x > 0 \\ [1,1][0,0]^T = 0 = |x| & ,x = 0 \\ [1,1][0,-x]^T = -x = |x| & ,x < 0 \end{cases}$$

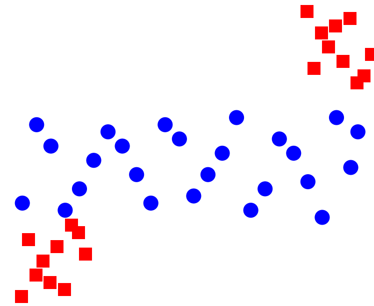
$$\Rightarrow f(x) = |x|, \forall x \in \mathbb{R}$$

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

6. **Two Clusters Walk Into a Bar. . . Part II: Electric Boogaloo** (15 points)



(a) Two thin vertical strips, side by side and touching.



(b) A wide blue horizontal band with two small red blobs at opposite corners, both labeled red.

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

Solution:

Both. The two strips are linearly separable by a vertical line, so K-Means with a vertical boundary and GMM (with two Gaussians whose means sit in middle of each strip) can both produce this assignment.

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

Solution:

Neither. K-Means can't assign two spatially disjoint red blobs to the same cluster while a blue cluster lies between them. It violates the definition of closest centroid since we have only 2 classes.

A standard GMM with one component per cluster also cannot place a single Gaussian's high-probability region over two disconnected red blobs separated by blue points without also claiming the blue dots in between.

(c) Which of the following is *not* a valid objective optimized at some step of K-Means or the EM algorithm for GMMs? Assume γ_{ik} denotes the soft responsibility of component k for point i, and $r_{ik} \in \{0, 1\}$ denotes a hard assignment.

A. $\sum_{i=1}^{n} \sum_{k=1}^{K} r_{ik} \|x_i - \mu_k\|^2$ (minimized over $\{r_{ik}\}$ and $\{\mu_k\}$)

B. $\sum_{i=1}^{n} \sum_{k=1}^{K} \gamma_{ik} \left[\log \pi_k + \log \mathcal{N}(x_i \mid \mu_k, \Sigma_k)\right]$ (maximized in the M-step)

C. $\sum_{i=1}^{n} \log\Big(\arg\max_k \pi_k \mathcal{N}(x_i \mid \mu_k, \Sigma_k)\Big)$ (maximized over $\{\pi_k, \mu_k, \Sigma_k\}$)

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

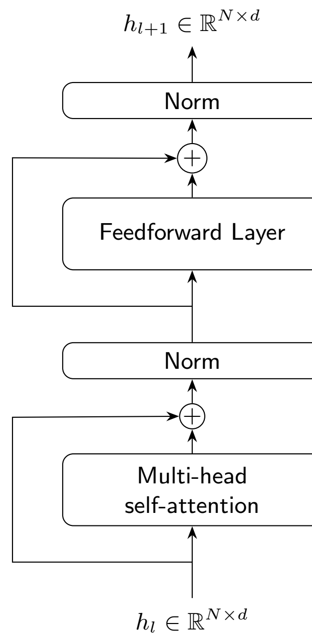
Solution:

(c) You should not take the log of an index value in likelihood maximization.

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

The goal of this question is to implement the core logic of a Transformer Encoder, the component of the Transformer model responsible for processing input sequences. A single encoder layer receives input from the previous layer $h_l \in \mathbb{R}^{N \times d}$ and produces $h_{l+1} \in \mathbb{R}^{N \times d}$. Here, N is the sequence length and d is the model dimension (or the hidden size). A Transformer Encoder has two main components:

- A multi-head self-attention module
- A fully-connected feed-forward layer



Answer the following questions based on the description above. Remember to check the cheatsheet if you need a reminder on the constructor or forward method signatures for various layers.

(a) Complete the code below for the feedforward layer shown in the figure. The feedforward layer takes an input $x \in \mathbb{R}^d$ and produces an output y as follows:

$$y_i = \sigma(W_i x + b_i)$$
$$y = W_o y_i + b_o$$

Here, $W_i, W_o \in \mathbb{R}^{d\times d}$, $b_i, b_o \in \mathbb{R}^d$, and σ is the Sigmoid function. Use `torch.nn.Linear` to implement this layer.

```
1  import torch
2  import torch.nn as nn
3  import torch.nn.functional as F
4
5  class FeedforwardLayer(nn.Module):
6      def __init__(self, d: int) -> None:
7          self.d = d
8          # Complete the init method
9
10
11
12
13
14
15
16
17
18
19     def forward(self, x: torch.Tensor) -> torch.Tensor:
20         # x: a tensor of shape (batch_size, N,  d)
21         # Complete the forward method
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44     # End implementation
```

Solution:

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class FeedforwardLayer(nn.Module):
    def __init__(self, d: int) -> None:
        self.d = d
        # Complete the init method
        self.w1 = nn.Linear(d, d, bias=True)
        self.w2 = nn.Linear(d, d, bias=True)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        # x: a tensor of shape (batch_size, N,  d)
        # Complete the forward method
        return self.w2(F.sigmoid(self.w1(x)))

    # End implementation
```

(b) Complete the code below to implement a Transformer Encoder based on the figure.

- Use `MultiheadAttention` to implement self-attention with `d` and `num_heads`.
- Use `torch.nn.LayerNorm` to implement the normalization.

```
1  import torch
2  import torch.nn as nn
3  import torch.nn.functional as F
4
5  # Assume that the following class implementation is provided.
6  # Do not implement this class and only use it as a reference
7  # to implement TransformerEncoder.
8  class MultiheadAttention(nn.Module):
9      def __init__(self, d: int, num_heads: int) -> None:
10         # Some implementation
11
12     def forward(self, key: torch.Tensor, value: torch.Tensor, query: torch.
       Tensor) -> torch.Tensor:
13         # key: (batch_size, N, d)
14         # value: (batch_size, N, d)
15         # query: (batch_size, N, d)
16         # Some implementation that takes key, value, and query to compute
17         # and return the attention output of shape (batch_size, N, d)
18         # as a torch.Tensor.
19
20 # Complete the implementation of the following class.
21 class TransformerEncoder(nn.Module):
22     def __init__(self, d: int, num_heads: int) -> None:
23         self.d = d
24         # Complete the init method
25
26
27
28
29
30
31
32
33     def forward(self, x: torch.Tensor) -> torch.Tensor:
34         # x: a tensor of shape (batch_size, N, d)
35         # Complete the forward method
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52     # End implementation
```

Solution:

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class TransformerEncoder(nn.Module):
    def __init__(self, d: int, num_heads: int) -> None:
        self.d = d
        self.attn = MultiheadAttention(
            embed_dim=d,
            num_heads=num_heads,
        )
        # Complete the init method
        self.norm1 = nn.LayerNorm(d)
        self.ff = FeedforwardLayer(d)
        self.norm2 = nn.LayerNorm(d)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        # x: a tensor of shape (batch_size, N, d)
        # Complete the forward method
        x = x + self.attn(key=x, value=x, query=x)
        x = self.norm1(x)
        x = x + self.ff(x)
        return self.norm2(x)

    # End implementation
```

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

```
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)
```

- **LayerNorm layer:** like batch normalization, but applies a normalization using all of the summed inputs to the neurons in a layer on a single training case $y = \frac{x-\mathbb{E}[x]}{\sqrt{\text{Var}[x]+\epsilon}} * \gamma + \beta$ where γ and β are trainable parameters:

```
torch.nn.LayerNorm(normalized_shape, eps=1e-05, elementwise_affine=True, bias=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: $J(C_1, \ldots, C_k, \mu_1, \ldots, \mu_k) = \Sigma_{k=1}^{K} \Sigma_{i \in C_k} ||x^{(i)} - \mu_k||^2$

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 $\mathcal{L}(x, c, l, g) = \frac{1}{N}(\mathcal{L}_{\text{cls}}(x, c) + \alpha \mathcal{L}_{\text{loc}}(x, l, g))$, 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.

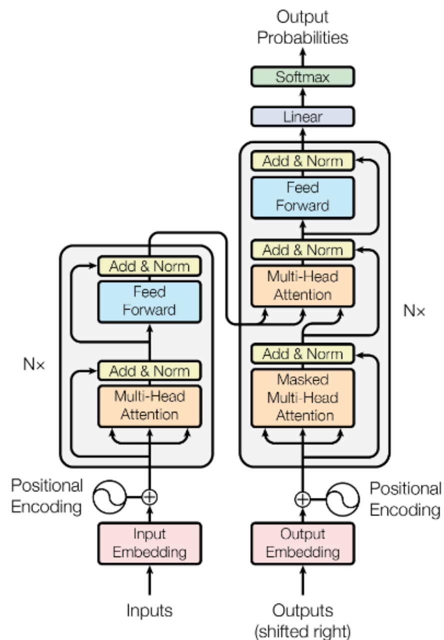


Figure 1: The Transformer - model architecture.

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:

```
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:

```
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
```