CS 189/289A, Final

Spring 2026

Name: _____

Email: _____@berkeley.edu

Student ID: _____

Room where you're taking the exam: _____

Name and SID of the person on your right: _____

Name and SID of the person on your left: _____

Instructions:

This exam consists of **68 points** spread out over **5 questions** and the Honor Code certification. The exam must be completed in **170 minutes** unless you have accommodations supported by a DSP letter.

Note that you should **select one choice** for questions with **circular bubbles**. There is always exactly one correct answer. Please **fully** shade in the circle to mark your answer. For questions with **square bubbles** you may select multiple options. There is always *atleast* one correct answer. Please **fully** shade in the square(s) to mark your answer. For all math questions, **please simplify your answer**. Please also **show your work** if a large box is provided.

You MUST write your Student ID number at the top of each page.

Honor Code [1 Pt]:

As a member of the UC Berkeley community, I act with honesty, integrity, and respect for others. I am the person whose name is on the exam, and I completed this exam in accordance with the Honor Code.

Signature: _____

This page has been intentionally left blank.

1 Machine Learning Trivia [8 Pts]

Select the best answer for each of the following questions.

(a) **[1 Pt]** Which of the following is the *best* characterization of the Universal Approximation Theorem (UAT)?

- ● A sufficiently wide single-hidden-layer network with a suitable activation can approximate any continuous function to arbitrary precision.
- ○ A sufficiently wide single-hidden-layer network with a suitable activation can approximate any continuous function to arbitrary precision, and gradient descent will find such an approximation.
- ○ A sufficiently deep network with a suitable activation can approximate any continuous function to arbitrary precision, but a single-hidden-layer network cannot.
- ○ A sufficiently wide single-hidden-layer network can approximate any function to arbitrary precision, with or without an activation function.

Explanation: The UAT is an *existence* result: given any continuous function and any $\epsilon > 0$, there exists a single-hidden-layer network that approximates it within ϵ. It does not guarantees learnability and it requires a nonlinear activation function.

(b) **[1 Pt]** Which of the following is **NOT** an advantage of convolutional layers over fully-connected layers for image processing?

- ○ Parameter efficiency through weight sharing across spatial positions.
- ○ Translation equivariance, so that a shifted input produces a correspondingly shifted feature map.
- ○ Preservation of spatial structure by operating on local neighborhoods rather than flattening the input.
- ● The ability to learn complex features without requiring non-linear activation functions between layers.

Explanation: Convolutional layers, like fully-connected layers, are linear operations. Without non-linear activations between layers, stacking multiple convolutional layers collapses to a single linear transformation, just as with fully-connected layers.

(c) **[1 Pt]** Which of the following best describes the role of pooling (e.g. max-pooling) layers in a CNN?

- ○ They apply learnable downsampling filters that adapt during training to select the most informative spatial regions.
- ○ They increase the spatial resolution of feature maps so that fine-grained details are preserved for later layers.

- (●) They reduce the spatial dimensions of feature maps, which grows the effective receptive field and introduces a degree of translation invariance.
- ( ) They normalize activations across spatial locations to stabilize training and prevent internal covariate shift.

Explanation: Pooling layers downsample feature maps by collapsing nearby activations, which reduces spatial dimensions, grows the effective receptive field of subsequent layers, and makes representations more robust to small spatial shifts. They have no learnable parameters.

(d) **[1 Pt]** In a Vision Transformer (ViT), how is an input image converted into a sequence that the transformer encoder can process?

- ◯ The image is passed through a stack of convolutional and pooling layers, and the final feature map is used as the token sequence.
- ◯ Each individual pixel is treated as a separate token in the sequence.
- ● The image is split into fixed-size non-overlapping patches, and each patch is linearly projected into a token embedding.
- ◯ A recurrent network scans the image row by row to produce a sequence of hidden states.

Explanation: ViT partitions the image into a grid of patches (e.g. 16×16), flattens each patch, and maps it through a learned linear projection to produce a sequence of token embeddings.

(e) **[1 Pt]** Suppose we are instruction-tuning a pre-trained language model using Supervised Fine-Tuning (SFT). How is the next-token prediction loss typically applied across a given `(prompt, response)` training pair?

- ◯ The loss is computed and averaged equally over all tokens in both the `prompt` and the `response`.
- ◯ The loss is computed only over the `prompt` tokens, effectively masking the `response`.
- ● The loss is computed only over the `response` tokens, effectively masking the `prompt` tokens.
- ◯ The loss applies a penalty term based on the KL divergence between the `prompt` and `response` distributions.

Explanation: The loss is computed only over the `response` tokens. Prompt tokens are masked out from the loss calculation so the model isn't penalized for failing to predict the user's prompt.

(f) **[1 Pt]** Suppose you perform Supervised Fine-Tuning (SFT) on a pre-trained language model. You consider two approaches: (1) *full-parameter SFT*, which updates all weights in the model, and (2) *output-head-only SFT*, which freezes the base model and updates weights of the output head only. After fine-tuning, the model is deployed for inference. Which of the following best describes the impact on **inference latency** compared to the original base model?

- ◯ Full-parameter SFT increases inference latency relative to the base model, but output-head-only SFT does not.
- ◯ Both approaches increase inference latency, but full-parameter SFT increases it more.
- ● Neither approach changes inference latency compared to the base model.

◯ Output-head-only SFT decreases inference latency because fewer parameters were updated during training.

Explanation: Neither approach changes inference latency. SFT updates the *values* of weight matrices but does not change the model architecture: the number of layers, the dimensions of each weight matrix, and the computation graph all remain identical.

(g) **[1 Pt]** What is one of the most important sources of information used by AlphaFold2 when predicting the structure of a protein?

- ● The sequences of proteins thought to be evolutionarily related to it
- ○ The structure of that same protein at its melting point
- ○ The name of the organism it came from
- ○ The sequences of proteins of the same length

Explanation: AlphaFold2 uses a transformer that operates over what's called a multiple sequence alignment (MSA) to learn covariation in residues across evoltuionarily related proteins, enabling it to develop a coarse contact map based on which residues appear to be interacting. This contact map serves as a prior for a separate structure module that can be thought of as refining the proposed structure to make it more biochemically plausible.

(h) **[1 Pt]** GEPA (Generalized Evolving Prompt Adaptation) is a method for improving LLM-based agents without modifying model weights. How does it work?

- ○ It fine-tunes a small adapter network on top of the frozen model using task-specific demonstrations.
- ○ It uses reinforcement learning to update the model's policy gradient estimates directly.
- ● It has the agent attempt a task, then uses a reflection model to suggest improvements to the agent's prompt based on the trajectory and reward.
- ○ It retrieves relevant few-shot examples from a database and prepends them to the prompt at inference time.

Explanation: GEPA keeps model weights frozen and instead optimizes the prompt. The agent produces a trajectory on a task, and a reflection model observes the trajectory and reward to suggest prompt updates. This makes it applicable to closed-weight models where SFT and RL are not possible.

2 Modeling Binary Data [18 Pts]

We observe n examples $\mathbf{x}_1, \ldots, \mathbf{x}_n$, where each example is a binary vector

$$\mathbf{x}_i = (x_{i1}, \ldots, x_{iD}) \in \{0,1\}^D.$$

Recall the Bernoulli probability mass function: for $x \in \{0,1\}$, $\text{Bern}(x \mid \mu) = \mu^x(1-\mu)^{1-x}$.

Mixture of Bernoulli distributions. Consider a Mixture of Bernoulli distributions (MoB) with K components. Each data point $\mathbf{x}_i$ is generated by first drawing a component assignment $z_i \in \{1, \ldots, K\}$ with

$$p(z_i = k) = \pi_k, \qquad 0 \le \pi_k \le 1, \qquad \sum_{k=1}^{K} \pi_k = 1,$$

and then drawing $\mathbf{x}_i$ conditionally on $z_i = k$:

$$p(\mathbf{x}_i \mid z_i = k) = \prod_{d=1}^{D} \mu_{kd}^{x_{id}} (1-\mu_{kd})^{1-x_{id}}, \qquad \mu_{kd} \in [0,1].$$

(a) **[2 Pts] Posterior responsibility**

For fixed MoB parameters (π, μ), define the posterior responsibility

$$r_{ik} := p(z_i = k \mid \mathbf{x}_i, \pi, \mu).$$

Fill in the boxes below:

$$r_{ik} = \frac{\boxed{\text{A}}}{\sum_{j=1}^{K} \boxed{\text{B}}}.$$

A should be in terms of π_k, μ_{kd}, and x_{id} , and **B** should be in terms of π_j, μ_{jd}, and x_{id}.

Solution: By Bayes' rule, $r_{ik} = \frac{p(z_i=k)\, p(\mathbf{x}_i \mid z_i=k)}{\sum_{j=1}^{K} p(z_i=j)\, p(\mathbf{x}_i \mid z_i=j)}$. Substituting:

$$\boxed{\text{A}} = \pi_k \prod_{d=1}^{D} \mu_{kd}^{x_{id}} (1-\mu_{kd})^{1-x_{id}}, \qquad \boxed{\text{B}} = \pi_j \prod_{d=1}^{D} \mu_{jd}^{x_{id}} (1-\mu_{jd})^{1-x_{id}}.$$

Bernoulli generative classifier. Now suppose each example $\mathbf{x}_i$ comes with a binary label $y_i \in \{0,1\}$. The *Bernoulli generative classifier* mirrors the MoB setup, but the observed label y_i takes the role of the latent assignment z_i:

$$p(y_i = 1) = \phi, \qquad p(y_i = 0) = 1 - \phi,$$

$$p(\mathbf{x}_i \mid y_i = c) = \prod_{d=1}^{D} \mu_{cd}^{x_{id}} (1 - \mu_{cd})^{1 - x_{id}}, \qquad c \in \{0, 1\}.$$

(b) **[4 Pts] From generative model to linear log-odds**

Applying Bayes' rule and substituting the Bernoulli factorization, one can show that the posterior log-odds of the generative classifier are

$$\log \frac{p(y = 1 \mid \mathbf{x})}{p(y = 0 \mid \mathbf{x})} = \log \frac{\phi}{1 - \phi} + \sum_{d=1}^{D} \left[x_d \log \frac{\mu_{1d}}{\mu_{0d}} + (1 - x_d) \log \frac{1 - \mu_{1d}}{1 - \mu_{0d}} \right]. \tag{1}$$

Prove that (1) is an affine function of $\mathbf{x}$, i.e., that there exist $\mathbf{w} \in \mathbb{R}^D$ and $b \in \mathbb{R}$ such that

$$\log \frac{p(y = 1 \mid \mathbf{x})}{p(y = 0 \mid \mathbf{x})} = \mathbf{w}^\top \mathbf{x} + b,$$

by stating the resulting w_d and b explicitly and clearly showing your algebraic steps.

Solution: Expanding the d-th summand:

$$\begin{aligned} x_d \log \frac{\mu_{1d}}{\mu_{0d}} + (1 - x_d) \log \frac{1 - \mu_{1d}}{1 - \mu_{0d}} &= x_d \log \frac{\mu_{1d}}{\mu_{0d}} + \log \frac{1 - \mu_{1d}}{1 - \mu_{0d}} - x_d \log \frac{1 - \mu_{1d}}{1 - \mu_{0d}} \\ &= x_d \left(\log \frac{\mu_{1d}}{\mu_{0d}} - \log \frac{1 - \mu_{1d}}{1 - \mu_{0d}} \right) + \log \frac{1 - \mu_{1d}}{1 - \mu_{0d}}. \end{aligned}$$

Combining the x_d-coefficient into a single log:

$$x_d \cdot \log \frac{\mu_{1d}(1 - \mu_{0d})}{\mu_{0d}(1 - \mu_{1d})} + \log \frac{1 - \mu_{1d}}{1 - \mu_{0d}}.$$

Summing over d and adding $\log \frac{\phi}{1-\phi}$:

$$\log \frac{p(y = 1 \mid \mathbf{x})}{p(y = 0 \mid \mathbf{x})} = \underbrace{\log \frac{\phi}{1 - \phi} + \sum_{d=1}^{D} \log \frac{1 - \mu_{1d}}{1 - \mu_{0d}}}_{b} + \sum_{d=1}^{D} x_d \underbrace{\log \frac{\mu_{1d}(1 - \mu_{0d})}{\mu_{0d}(1 - \mu_{1d})}}_{w_d}.$$

So:

$$w_d = \log \frac{\mu_{1d}(1 - \mu_{0d})}{\mu_{0d}(1 - \mu_{1d})}, \qquad b = \log \frac{\phi}{1 - \phi} + \sum_{d=1}^{D} \log \frac{1 - \mu_{1d}}{1 - \mu_{0d}}.$$

Logistic regression takes the same linear log-odds form as its starting point, but fits $\mathbf{w}$ and b directly to data rather than deriving them from (ϕ, μ):

$$p(y = 1 \mid \mathbf{x}) = \sigma(\mathbf{w}^\top \mathbf{x} + b).$$

(c) **[2 Pts] Deriving cross-entropy loss**

Given a training set $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$ with $y_i \in \{0, 1\}$, let $\hat{y}_i := \sigma(\mathbf{w}^\top \mathbf{x}_i + b) = p(y_i = 1 \mid \mathbf{x}_i)$. Note that $p(y_i \mid \mathbf{x}_i)$ is a Bernoulli distribution with parameter $\hat{y}_i$.

Derive the negative log-likelihood, also known as the *binary cross-entropy* loss. Fill in the box below:

$$\mathcal{L}(\mathbf{w}, b) = -\frac{1}{n} \sum_{i=1}^n \log p(y_i \mid \mathbf{x}_i) = -\frac{1}{n} \sum_{i=1}^n \Big[\boxed{\text{C}} \Big].$$

C should involve only y_i, $\hat{y}_i$, and logarithms.

Solution: Since $y_i \mid \mathbf{x}_i$ is Bernoulli with parameter $\hat{y}_i$, we have $p(y_i \mid \mathbf{x}_i) = \hat{y}_i^{y_i}(1 - \hat{y}_i)^{1-y_i}$. Taking $-\log$:

$$\boxed{\text{C}} = y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i).$$

(d) **Linearly separable training data**

Suppose the training data is *linearly separable*: there exist $\mathbf{w}^*, b^*$ such that $\mathbf{w}^{*\top} \mathbf{x}_i + b^* > 0$ for all i with $y_i = 1$ and $\mathbf{w}^{*\top} \mathbf{x}_i + b^* < 0$ for all i with $y_i = 0$.

Consider scaling these parameters by $\alpha > 0$, i.e., using $(\alpha \mathbf{w}^*, \alpha b^*)$.

(i) **[1 Pt]** As $\alpha \to \infty$, what does the loss $\mathcal{L}(\alpha \mathbf{w}^*, \alpha b^*)$ converge to?

- ● 0
- ○ 1
- ○ $+\infty$
- ○ $-\infty$

Explanation: For each i with $y_i = 1$, scaling gives $\alpha(\mathbf{w}^{*\top} \mathbf{x}_i + b^*) \to +\infty$, so $\hat{y}_i = \sigma(\cdot) \to 1$ and $-\log \hat{y}_i \to 0$. Similarly for $y_i = 0$: $\hat{y}_i \to 0$ and $-\log(1 - \hat{y}_i) \to 0$. Every term in the sum vanishes, so $\mathcal{L} \to 0$.

(ii) **[1 Pt]** Which of the following **most directly** follows from this observation when training on *linearly separable data*?

- ○ The cross-entropy loss is non-convex for linearly separable data, so gradient descent converges to a local minimum.
- ○ Gradient clipping is needed to prevent the training loss from diverging.

- ● Without regularization, gradient descent on linearly separable data will drive $\|\mathbf{w}\| \to \infty$.
- ○ The model requires careful initialization near the separating hyperplane to avoid divergence.

Explanation: Since the infimum of the loss (zero) is never attained at any finite point, gradient descent can always find a direction that decreases the loss further by increasing $\|\mathbf{w}\|$. This drives $\|\mathbf{w}\| \to \infty$; the MLE does not exist. Regularization (e.g., adding $\lambda\|\mathbf{w}\|^2$ to the loss) resolves this by penalizing large weights.

(e) **Comparing the Bernoulli generative classifier with logistic regression**

(i) **[1 Pt]** How many trainable parameters does the *Bernoulli generative classifier* from part (b) have?

○ D ○ $D+1$ ○ $2D$ ● $2D+1$

Explanation: The generative classifier estimates ϕ (1 parameter) plus μ_{0d} and μ_{1d} for each of D coordinates ($2D$ parameters), totaling $2D+1$.

(ii) **[1 Pt]** How many trainable parameters does *logistic regression* have?

○ D ● $D+1$ ○ $2D$ ○ $2D+1$

Explanation: Logistic regression has $\mathbf{w} \in \mathbb{R}^D$ and $b \in \mathbb{R}$, totaling $D+1$.

(iii) **[1 Pt]** Specifying ϕ and all μ_{cd} determines a posterior $p(y=1 \mid \mathbf{x})$ that can be written in the same functional form as logistic regression:

$$p(y=1 \mid \mathbf{x}) = \sigma(\mathbf{w}^\top \mathbf{x} + b)$$

for suitable $\mathbf{w}, b$.

● True ○ False

Explanation: By part (b), specifying (ϕ, μ) yields $\log \frac{p(y=1|\mathbf{x})}{p(y=0|\mathbf{x})} = \mathbf{w}^\top \mathbf{x} + b$ for the derived $\mathbf{w}, b$. Exponentiating and solving shows $p(y=1 \mid \mathbf{x}) = \sigma(\mathbf{w}^\top \mathbf{x} + b)$.

(iv) **[1 Pt]** For every choice of $w \in \mathbb{R}^D$ and $b \in \mathbb{R}$, there is a unique choice of ϕ and μ_{cd} that gives exactly the same posterior $p(y=1 \mid x)$ for all $x \in \{0,1\}^D$.

○ True ● False

Explanation: Consider the following counterexample ($D=1$): The posterior depends on $(\phi, \mu_{10}, \mu_{00})$ only through $w_1 = \text{logit}(\mu_{10}) - \text{logit}(\mu_{00})$ and $b = \text{logit}(\phi) + \log \frac{1-\mu_{10}}{1-\mu_{00}}$. Take $w_1 = 0,\ b = 0$. Then $\mu_{10} = \mu_{00}$ (any shared value in $(0,1)$) and $\phi = \frac{1}{2}$. Both $(\frac{1}{2}, 0.3, 0.3)$ and $(\frac{1}{2}, 0.7, 0.7)$ yield $w_1 = 0,\ b = 0$ and therefore the exact same posterior, so the choice is not unique.

(v) **[1 Pt]** The Bernoulli generative classifier has closed-form MLEs for ϕ and the μ_{cd}, while logistic regression generally does not have a closed-form MLE for $\mathbf{w}, b$.

● True ○ False

Explanation: The generative MLEs are $\hat{\phi} = \frac{1}{n}\sum_i y_i$ and $\hat{\mu}_{cd} = \frac{\sum_{i:y_i=c} x_{id}}{\sum_i \mathbf{1}\{y_i=c\}}$ — all closed-form. Logistic regression has a concave log-likelihood with no closed-form maximizer and requires iterative optimization (e.g., gradient descent or Newton's method).

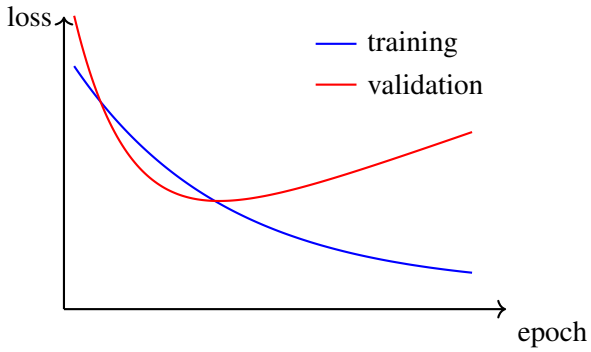
(vi) **[1 Pt]** Both the Bernoulli generative classifier and logistic regression produce linear decision boundaries in the original feature vector $\mathbf{x}$.

● True ○ False

Explanation: The generative classifier's log-odds are linear in $\mathbf{x}$ by part (b)(iii), giving a linear decision boundary $\{\mathbf{x} : \mathbf{w}^\top \mathbf{x} + b = 0\}$. Logistic regression's decision boundary is also $\{\mathbf{x} : \mathbf{w}^\top \mathbf{x} + b = 0\}$ by construction.

(f) **Choosing what to do in practice**

(i) **[1 Pt]** You train logistic regression with gradient descent. As training proceeds, the training loss continues to decrease, but the validation loss begins to increase:

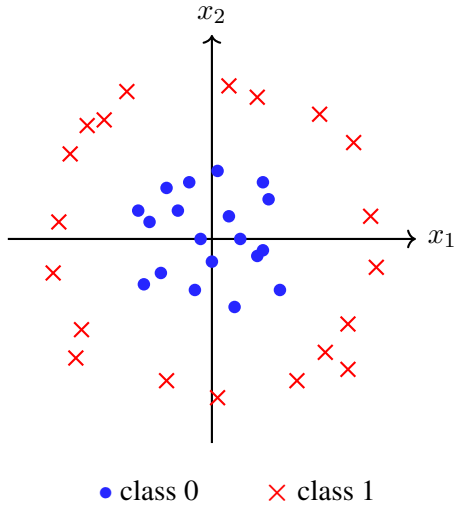


Which change is *most likely* to help?

- ● Re-train using a stronger weight-decay penalty
- ○ Re-train logistic regression after augmenting the input with pairwise product features $x_i x_j$
- ○ Re-train logistic regression using a larger learning rate
- ○ Re-training using mean-squared error (MSE) instead of cross-entropy as your loss function.

Explanation: The divergence between training and validation loss is the classic signature of overfitting. Weight decay directly combats this by penalizing large weights and reducing effective model capacity.

(ii) **[1 Pt]** The figure below shows a 2D training set with two classes.



Logistic regression trained on the original two input features gives a straight-line decision boundary and has high training error. Which change is *most likely* to help?

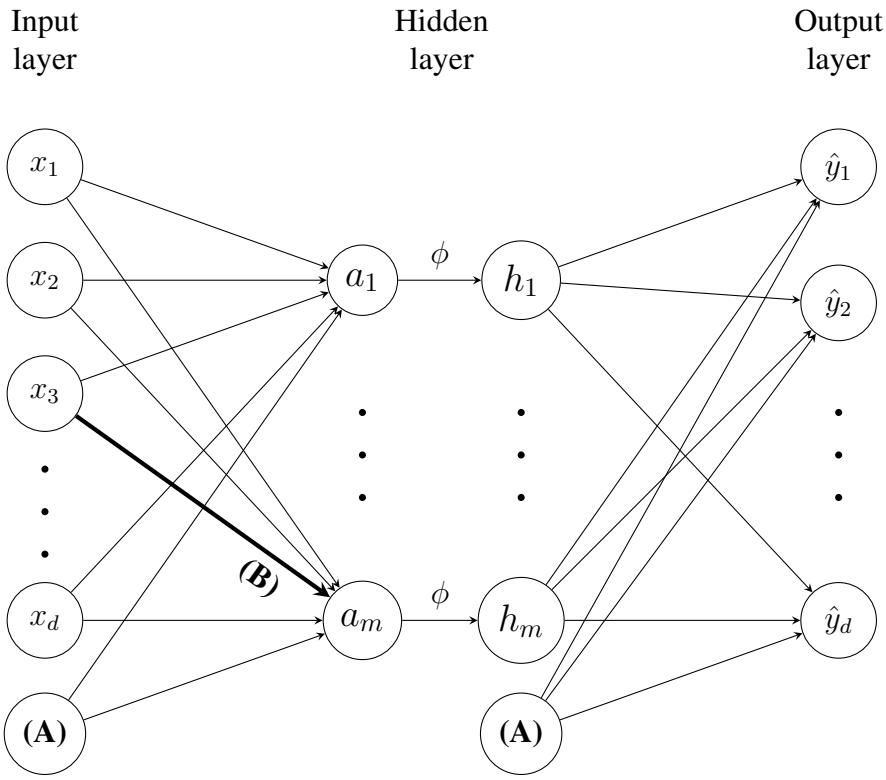
- (●) Re-train after augmenting the input with the features x_1^2 and x_2^2
- ( ) Replace the sigmoid function in logistic regression with the function $f(x) = \frac{x^2}{1+x^2}$
- ( ) Re-train after augmenting the input with the feature $x_1 x_2$
- ( ) Replace logistic regression with the Bernoulli generative classifier

Explanation: Logistic regression's decision boundary is linear in its input features. A linear boundary in the augmented feature space (x_1, x_2, x_1^2, x_2^2) corresponds to a *elliptical* boundary in the original (x_1, x_2) space, which can capture the circular separation visible in the plot.

3 Mirror Neural Network [16 pts]

Consider the following neural network with input $\mathbf{x} \in \mathbb{R}^d$, a single hidden layer of width m, and output $\hat{\mathbf{y}} \in \mathbb{R}^d$. Crucially, the same weight matrix $\mathbf{W} \in \mathbb{R}^{m \times d}$ is used in both layers: $\mathbf{W}$ maps from input to hidden, and $\mathbf{W}^\top$ maps from hidden to output. We use the notation $W_{i,j} \in \mathbb{R}$ to refer to the scalar at the i-th row and j-th column.

Each layer also has its own **bias vector** — $\mathbf{b}^{(1)} \in \mathbb{R}^m$ for the hidden layer, and $\mathbf{b}^{(2)} \in \mathbb{R}^d$ for the output layer.



(a) **Network Fundamentals**

i. **[1 Pt]** What value should appear inside the nodes labeled **(A)**?

- ● 1
- ○ 0
- ○ ϕ
- ○ w_{m+1} (a learnable scalar parameter)

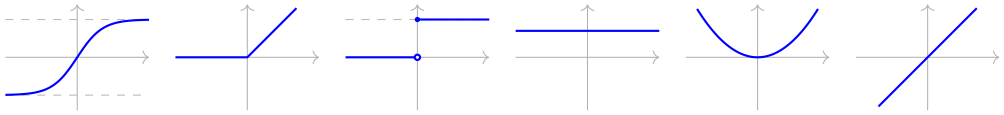
Explanation: The nodes labeled (A) are bias nodes. They feed a constant 1 into each layer, which gets scaled by the bias vector (b_1 in the hidden layer, b_2 in the output layer), providing a learnable offset to each unit's pre-activation.

ii. **[1 Pt]** The bold arrow labeled **(B)** connects input x_3 to hidden unit m. What is the scalar weight on this arrow?

● $W_{m,3}$ ○ $W_{3,m}$ ○ $W^{\top}_{3m,m}$ ○ $b^{(1)}_3$

Explanation: Row m of $\mathbf{W}$ contains the weights into hidden unit m, and column 3 corresponds to input x_3, so the weight is W_{m3}.

iii. **[2 Pts]** The activation function ϕ in the hidden layer must satisfy certain properties for the network to be trainable via gradient descent and to be more expressive than a linear model.
Which of the following are valid choices for ϕ? **Select all that apply.**

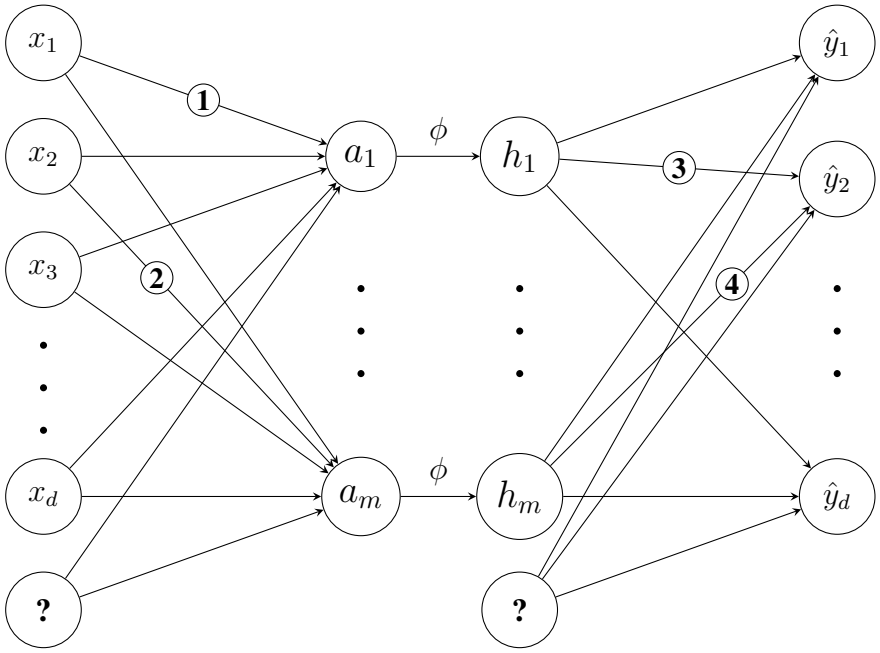


■ **(i)** ■ **(ii)** ☐ **(iii)** ☐ **(iv)** ■ **(v)** ☐ **(vi)**

Explanation: A valid activation must be (1) nonlinear, so the network is more expressive than a linear model, and (2) have a usable gradient almost everywhere for training. (Note that while the gradient is technically undefined at 0 for (ii), the ReLU function, in practice we can just choose to define it as 0).

(b) **[1 Pt] Mirror weights**

Four arrows in the diagram below are marked with circled numbers ①, ②, ③, and ④. Because the Mirror Network reuses the same weight matrix W in both layers, some pairs of arrows correspond to the **same scalar weight**.



Which of the following pairs of arrows correspond to the same scalar weight?

○ ① and ② ○ ② and ③ ○ ③ and ④
○ ① and ③ ● ② and ④ ○ ① and ④

Explanation: Arrow ① goes from x_1 to hidden unit 1, so it carries weight W_{11}. Arrow ② goes from x_2 to hidden unit m, so it carries weight W_{m2}. Arrow ③ goes from h_1 to $\hat{y}_2$, so it carries weight $(\mathbf{W}^\top)_{21} = W_{12}$. Arrow ④ goes from h_m to $\hat{y}_2$, so it carries weight $(\mathbf{W}^\top)_{2m} = W_{m2}$. Thus ② and ④ both equal W_{m2}.

For the remaining parts, assume the network uses the sigmoid activation $\phi = \sigma$ where σ is the logistic sigmoid. Given a target $\mathbf{t} \in \mathbb{R}^d$, the loss is

$$L = \tfrac{1}{2} \|\hat{\mathbf{y}} - \mathbf{t}\|_2^2.$$

(c) **[4 Pts] Backpropagation**

Compute $\frac{\partial L}{\partial \mathbf{b}_k^{(1)}}$ for a single bias parameter $\mathbf{b}_k^{(1)}$, for $k \in \{1, \ldots, m\}$. Show your work by clearly identifying each step of the chain rule. You may leave the sigmoid derivative as $\sigma'(\cdot)$ and use $\mathbf{w}_k^\top$ to denote the k-th row of $\mathbf{W}$ (equivalently, the k-th column of $\mathbf{W}^\top$, written as a column vector).

Solution: We apply the chain rule along the path $L \to \hat{\mathbf{y}} \to h_k \to a_k \to \mathbf{b}_k^{(1)}$:

$$\frac{\partial L}{\partial \mathbf{b}_k^{(1)}} = \frac{\partial L}{\partial \hat{\mathbf{y}}} \cdot \frac{\partial \hat{\mathbf{y}}}{\partial h_k} \cdot \frac{\partial h_k}{\partial a_k} \cdot \frac{\partial a_k}{\partial \mathbf{b}_k^{(1)}}$$

Computing each term:

1. $\frac{\partial L}{\partial \hat{\mathbf{y}}} = (\hat{\mathbf{y}} - \mathbf{t})^\top \in \mathbb{R}^{1 \times d}$
2. $\frac{\partial \hat{\mathbf{y}}}{\partial h_k} = \mathbf{w}_k \in \mathbb{R}^d$, since $\hat{\mathbf{y}} = \mathbf{W}^\top \mathbf{h} + \mathbf{b}^{(2)}$ and $\mathbf{w}_k$ is the k-th column of $\mathbf{W}^\top$. f
3. $\frac{\partial h_k}{\partial a_k} = \sigma'(a_k) \in \mathbb{R}$, since $h_k = \sigma(a_k)$.
4. $\frac{\partial a_k}{\partial \mathbf{b}_k^{(1)}} = 1$, since $a_k = \mathbf{w}_k^\top \mathbf{x} + \mathbf{b}_k^{(1)}$.

Combining:

$$\boxed{\frac{\partial L}{\partial \mathbf{b}_k^{(1)}} = (\hat{\mathbf{y}} - \mathbf{t})^\top \mathbf{w}_k \cdot \sigma'(a_k)}$$

(d) **[2 Pts] Learnable parameters**

How many learnable parameters does this network have?

- ◯ $2md + m + d$
- ● $md + m + d$
- ◯ $md + m$
- ◯ md
- ◯ $2md + 2m + 2d$
- ◯ $md + 2m + 2d$

Explanation: The learnable parameters are $\mathbf{W} \in \mathbb{R}^{m \times d}$ (md entries), $\mathbf{b}^{(1)} \in \mathbb{R}^m$ (m entries), and $\mathbf{b}^{(2)} \in \mathbb{R}^d$ (d entries). Since the output layer reuses $\mathbf{W}^\top$, no second weight matrix is introduced. This gives $md + m + d$.

Suppose for that $m < d$. We can view the Mirror Network as an **encoder-decoder**:

- The **encoder** $\mathcal{E}(\mathbf{x}) = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}^{(1)})$ encodes an input $\mathbf{x} \in \mathbb{R}_d$ into a latent $\mathbf{h} \in \mathbb{R}^m$.
- The **decoder** $\mathcal{D}(\mathbf{z}) = \mathbf{W}^\top \mathbf{h} + \mathbf{b}^{(2)}$ decodes a latent $\mathbf{h} \in \mathbb{R}^m$ to output $\hat{\mathbf{y}} \in \mathbb{R}^d$.

(e) **[2 Pts] Weight sharing** Which of the following are valid reasons why weight-sharing between the encoder and decoder might be useful? **Select all that apply.**

- ■ It halves the number of weight parameters compared to using independent encoder and decoder matrices.
- ☐ It ensures that the representation $R(\mathbf{x})$ is a lossless encoding of $\mathbf{x}$, since the decoder can always invert it.
- ■ It constrains the encoder and decoder to share geometric structure, acting as a form of regularization.
- ☐ It guarantees that the gradient of the loss with respect to $\mathbf{W}$ from the encoder layer equals the gradient with respect to $\mathbf{W}$ from the decoder layer.

Explanation: The first and third options are correct. Sharing $\mathbf{W}$ reduces the weight parameters from $2md$ to md, and it restricts the hypothesis class by forcing the encoder and decoder to share geometric structure, which acts as implicit regularization.

The second option is false: when $m < d$, the projection through $\mathbf{W}$ is a dimensionality bottleneck that discards information. Even when $m \geq d$, the decoder $\mathbf{W}^\top$ is constrained to be the transpose of the encoder, not its inverse. The fourth option is false: weight tying means the encoder and decoder contributions to the gradient *sum* into a single $\nabla_{\mathbf{W}} \mathcal{L}$, but the individual per-layer contributions are generally not equal.

(f) **[2 Pts] Batch normalization** Suppose we add a batch normalization layer between the pre-activation $\mathbf{a}$ and the activation σ, with learnable shift $\boldsymbol{\beta} \in \mathbb{R}^m$ and scale $\boldsymbol{\gamma} \in \mathbb{R}^m$. We denote the batch mean as $\boldsymbol{\mu} \in \mathbb{R}^m$ and the batch variance as $\boldsymbol{\nu}^2 \in \mathbb{R}^m$. Which of the following are true? **Select all that apply.**

- ■ Batch normalization introduces $2m$ new learnable parameters per layer.
- ☐ Batch normalization introduces $4m$ new learnable parameters per layer, since $\boldsymbol{\mu}$ and $\boldsymbol{\nu}^2$ must also be learned.

- ☐ At test time, $\boldsymbol{\mu}$ and $\boldsymbol{\nu}^2$ are recomputed from each test input on the fly.
- ■ At test time, $\boldsymbol{\mu}$ and $\boldsymbol{\nu}^2$ are replaced by running averages collected during training.
- ☐ At test time, batch normalization is skipped entirely and the pre-activation $\mathbf{a}$ is passed directly into σ.

Explanation: Only $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ are learned (m parameters each, $2m$ total). The batch mean $\boldsymbol{\mu}$ and variance $\boldsymbol{\nu}^2$ are computed from data during training, not learned via gradient descent. At test time, since there is no batch to compute statistics over, $\boldsymbol{\mu}$ and $\boldsymbol{\nu}^2$ are replaced by exponential moving averages accumulated during training. Batch normalization is not skipped at test time; it still normalizes using these stored statistics and applies the learned $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$.

(g) **[1 Pt] Bias redundancy under batch normalization** A classmate claims that when batch normalization is applied to the pre-activation $\mathbf{a} = \mathbf{W}\mathbf{x} + \mathbf{b}^{(1)}$, the bias $\mathbf{b}^{(1)}$ becomes redundant and can be removed without changing the function the network can express. Do you agree?

- ◯ Disagree: without $\mathbf{b}^{(1)}$, the pre-activation is forced to have zero mean, restricting the network's expressiveness.
- ◯ Disagree: $\mathbf{b}^{(1)}$ is needed to break the symmetry of $\mathbf{W}$ during training.
- ● Agree: $\mathbf{b}^{(1)}$ shifts every pre-activation by a constant, which shifts the batch mean $\boldsymbol{\mu}$ by the same amount; the centering step cancels $\mathbf{b}^{(1)}$ exactly, and the learnable shift $\boldsymbol{\beta}$ takes over the role of the bias.
- ◯ Agree, but only when $\boldsymbol{\gamma} = \mathbf{1}$; for other values of $\boldsymbol{\gamma}$, removing $\mathbf{b}^{(1)}$ changes the network's output because the scale parameter interacts with the bias before centering occurs.

Explanation: Adding $\mathbf{b}^{(1)}$ shifts every pre-activation $a^{(i)}$ by a constant, which shifts $\boldsymbol{\mu}$ by the same amount. The centering step $a^{(i)} - \boldsymbol{\mu}$ cancels $\mathbf{b}^{(1)}$ exactly, regardless of $\boldsymbol{\gamma}$. The learnable shift $\boldsymbol{\beta}$ then serves as the bias, making $\mathbf{b}^{(1)}$ unnecessary.

4 CNNs and Self-Supervised Learning [11 pts]

(a) **Convolutional layers**

i. **[1 Pt]** An input tensor has shape $3 \times 24 \times 24$ (channels $\times$ height $\times$ width). After applying a convolutional layer with 16 filters, kernel size 7×7, stride 1, and no padding, what is the shape of the output tensor?

- ● $16 \times 18 \times 18$
- ○ $16 \times 30 \times 30$
- ○ $3 \times 30 \times 30$
- ○ $16 \times 24 \times 24$

Explanation: The number of output channels equals the number of filters (16). With no padding, each spatial dimension shrinks by $k - 1 = 6$, giving $24 - 6 = 18$.

ii. **[1 Pt]** An input tensor has shape $8 \times 20 \times 20$. After applying a 2×2 max-pooling layer with stride 2, what is the shape of the output tensor?

- ○ $4 \times 10 \times 10$
- ○ $8 \times 20 \times 20$
- ● $8 \times 10 \times 10$
- ○ $8 \times 9 \times 9$

Explanation: Max-pooling with kernel 2 and stride 2 halves each spatial dimension ($20 \to 10$) but does not change the number of channels.

iii. **[1 Pt]** An input tensor has shape $16 \times 14 \times 14$. After applying a convolutional layer with 32 filters, kernel size 3×3, stride 2, and padding 1, what is the shape of the output tensor?

- ○ $32 \times 14 \times 14$
- ○ $32 \times 6 \times 6$
- ● $32 \times 7 \times 7$
- ○ $16 \times 7 \times 7$

Explanation: With padding 1, kernel size 3, and stride 2, the spatial dimensions become $\lfloor (14 - 3 + 2)/2 \rfloor + 1 = 7$. The number of channels becomes 32.

iv. **[1 Pt]** The following 3×3 kernel is applied to a single-channel grayscale image (pixel values are nonnegative):

$$\mathbf{K} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

Which of the following best describes the behavior of this kernel?

- ◯ It averages nearby pixels, producing a smoothed version of the image where fine details are suppressed.
- ◯ It outputs a large positive value in constant-intensity regions and zero at sharp transitions.
- ● It outputs zero in constant-intensity regions and a large value at pixels that differ significantly from their neighbors.
- ◯ It shifts the image by one pixel in the horizontal direction.

Explanation: The kernel weights sum to zero ($4 - 4 = 0$), so any constant region produces an output of exactly zero. At a pixel that differs from its neighbors, the center weight of 4 amplifies the center while the -1 weights subtract the neighbors, producing a strong response. This is also known as a Laplacian kernel.

(b) **[1 Pt] Pretext tasks**

A student trains a CNN on a large unlabeled dataset of animal images. During training, each image is randomly rotated by one of four angles:

$$0^\circ, \quad 90^\circ, \quad 180^\circ, \quad 270^\circ.$$

The model is trained to predict which rotation was applied. After this training, the student fine-tunes the CNN on a small labeled dataset to distinguish cats from dogs. Which statement identifies pretext task(s) in this setup?

- ◯ The pretext task is cat-vs-dog classification.
- ● The pretext task is rotation prediction.
- ◯ Both cat-vs-dog classification and rotation prediction are considered pretext tasks.
- ◯ Neither cat-vs-dog classification nor rotation prediction are considered pretext tasks.

Explanation: The model generates its own labels by rotating images and recording the applied angle, making rotation prediction a self-supervised pretext task. The later cat-vs-dog problem is the downstream task.

(c) **[1 Pt] Transfer learning with CNNs**

You have a CNN pretrained on ImageNet and only 200 labeled examples for a new bird classification task. What is the most reasonable transfer-learning strategy?

- ◯ Train every layer from scratch on the 200 examples.
- ● Freeze the pretrained convolutional layers and fine-tune only the last (classifier) layer.
- ◯ Fine-tune all layers with a high learning rate to adapt quickly to the new task.
- ◯ Freeze all layers and use the pretrained network's outputs directly without any retraining.

Explanation: With only 200 labeled examples, training from scratch or aggressively fine-tuning all layers is likely to overfit. The pretrained convolutional layers already extract useful visual features from ImageNet, so freezing them and replacing the final classifier head is the most data-efficient strategy. Using the network with no retraining at all would fail because the original classifier predicts ImageNet classes, not bird species.

(d) **Convolutional autoencoders**

i. **[1 Pt]** A convolutional autoencoder maps an input image $\mathbf{x}$ to a latent code $\mathbf{z} = f(\mathbf{x})$, then reconstructs the image as $\hat{\mathbf{x}} = g(\mathbf{z})$. The model is trained using a reconstruction loss such as $\|g(f(\mathbf{x})) - \mathbf{x}\|_2^2$. Assume that the dimensions of the latent code are smaller than the dimensions of the input image. What is the main purpose of the latent bottleneck $\mathbf{z}$?

- ● To force a compressed representation that captures the most important structure in the input.
- ○ To introduce stochasticity into the encoding, which acts as regularization during training.
- ○ To ensure the reconstruction loss converges to zero.
- ○ To make the decoder's job easier by providing a lower-dimensional input to reconstruct from.

Explanation: The bottleneck restricts how much information passes from encoder to decoder, forcing the encoder to learn a compact representation that preserves the most important structure in the data.

ii. **[1 Pt]** Suppose an autoencoder for 28×28 MNIST images uses a latent code $\mathbf{z} \in \mathbb{R}^k$. What is the most likely effect of making k extremely small (e.g. $k = 1$)?

- ● Reconstructions become blurry or inaccurate because the bottleneck cannot preserve enough information about the input.
- ○ Reconstructions remain sharp, but the model takes much longer to converge during training.
- ○ The encoder learns to ignore the input and the decoder memorizes the training set.
- ○ The model fails to train entirely because gradients cannot flow through a one-dimensional bottleneck.

Explanation: A very small bottleneck forces the encoder to discard most information about the input, resulting in poor reconstructions. Gradients still flow through the bottleneck regardless of its size, and the encoder cannot be bypassed since the decoder only receives $\mathbf{z}$.

(e) **[2 Pts] Training a text-conditioned heatmap model**

A student wants to train a model that takes as input an image $\mathbf{x}$ and a text query q (e.g. $q =$ "red cube") and outputs a heatmap

$$H_q(\mathbf{x}) \in [0,1]^{h \times w},$$

where large values indicate pixels relevant to the text query. The student has a dataset of images paired with text queries and pixel-level binary masks $M_q(\mathbf{x}) \in \{0,1\}^{h \times w}$ marking the queried object. Which training strategy is most appropriate?

- ○ Train an image encoder that produces a single global feature vector for the image, concatenate it with the text embedding, and predict the heatmap with a fully connected layer supervised by the ground-truth mask.
- ● Train an image encoder that produces spatial image features, train a text encoder that embeds the query, compute a similarity score between the query embedding and each spatial feature, and supervise the resulting heatmap against the ground-truth mask.
- ○ Train an image encoder that produces spatial image features, train a text encoder that embeds the query, compute a similarity score between the query embedding and each spatial feature, and supervise the resulting heatmap using a reconstruction loss that decodes the heatmap back into the original image.
- ○ Train an image encoder and text encoder jointly using only a contrastive loss between global image and text embeddings, then extract the heatmap from the attention weights at test time without any mask supervision.

Explanation: The model should produce *spatial* image features rather than only a single global image embedding. For example, let

$$\mathbf{F}(x) \in \mathbb{R}^{h \times w \times d}$$

be a grid of patch or pixel features from an image encoder, and let

$$\mathbf{t}(q) \in \mathbb{R}^d$$

be the text embedding from a text encoder. A heatmap can be produced by comparing the text embedding to each spatial image feature:

$$S_{ij} = \mathrm{sim}(\mathbf{F}_{ij}(x), \mathbf{t}(q)),$$

where sim could be cosine similarity or a dot product. Then the predicted heatmap could be

$$H_q(x)_{ij} = \sigma(S_{ij}),$$

where σ is the sigmoid function.

Since the dataset includes pixel-level masks $M_q(x)$, the model can be trained with a dense binary cross-entropy loss:

$$\mathcal{L}_{\text{mask}} = -\sum_{i,j} \left[M_q(x)_{ij} \log H_q(x)_{ij} + (1 - M_q(x)_{ij}) \log(1 - H_q(x)_{ij}) \right].$$

This teaches the model to assign high scores to pixels or patches relevant to the query and low scores elsewhere.

This is related to CLIP-style image-text representation learning, except that instead of matching one global image embedding to one text embedding, the model matches a text embedding to many spatial image features. This makes the model suitable for dense prediction tasks such as grounding, segmentation, and relevance heatmap prediction.

(f) **[1 Pt] Positive and negative pairs**

In contrastive learning, an image $\mathbf{x}$ is used as a reference. Which of the following is the best example of a positive pair?

- ◯ $\mathbf{x}$ and the next image in the dataset.
- ● $\mathbf{x}$ and a color-distorted version of $\mathbf{x}$.
- ◯ $\mathbf{x}$ and the pixel-nearest neighbor from another class.
- ◯ $\mathbf{x}$ and a random noise image.

Explanation: A positive pair should preserve the semantic content of $\mathbf{x}$. Data augmentations such as cropping, color distortion, and blur create views that look different at the pixel level but represent the same underlying content, making them effective positive examples.

(g) **[1 Pt] Zero-shot classification with CLIP**

A CLIP model has a jointly trained image encoder and text encoder. You want to classify an image as one of "cat," "dog," or "plane" without training a new classifier. What should you do?

- ◯ Encode the image, then pass its embedding through a linear classifier trained on the three class labels.
- ● Encode the image and each class label as text, then choose the label whose text embedding has the highest cosine similarity to the image embedding.
- ◯ Encode each class label as text and use the text embeddings as cluster centroids, then assign the image to the nearest cluster based on pixel distance.
- ◯ Encode the image three times with different random augmentations, average the embeddings, and pick the class whose text embedding is closest to the average.

Explanation: CLIP aligns image and text embeddings in a shared space. For zero-shot classification, each candidate class is converted into a text prompt (e.g. "a photo of a cat"), embedded with the text encoder, and compared to the image embedding via cosine similarity. The class with the highest similarity is the prediction. No additional training is needed.

(h) **[1 Pt] CLIP for robotics**

Suppose we train a model using the CLIP contrastive objective on a dataset of image-action pairs. A robot must use this model at test time to select an action given the current camera image. What is the *key* limitation? Assume actions are represented as real-valued vectors.

- ◯ The contrastive objective cannot be applied to image-action pairs.
- ◯ The model would not be able to map images and actions to a shared embedding space.
- ● The model can score how well a given action matches an image, but cannot generate or sample actions conditioned on an image.
- ◯ The model would require text descriptions of actions as an intermediate step at test time.

Explanation: CLIP-style contrastive training produces a discriminative model: given an image and an action, it can score their compatibility, but it does not model the conditional distribution $p(\mathbf{a} \mid \mathbf{x})$ needed to produce actions. This is the same reason CLIP alone cannot generate image captions: it can rank candidate texts but has no mechanism to synthesize new ones.

5 Transformers [15 Pts]

Transformers are the dominant class of language models. Taking a sequence of token embeddings as input, they repeatedly apply the attention and feed-forward layers to the processed vectors. If set up properly, this model enables generating tokens of new text or images.

(a) **[1 Pt]** Which loss function would be most effective for training a transformer to perform next-token prediction?

- ◯ Mean Squared Error (MSE)
- ◯ InfoNCE (Contrastive Loss)
- ◯ Attention Loss
- ● Cross-Entropy Loss

Explanation: Note that next-token prediction is fundamentally a classification problem: the model is tasked with picking which token out of a discrete, finite vocabulary is best suited to follow the inputted tokens. Therefore, we use a Cross-Entropy Loss.

(b) **[1 Pt]** Each transformer block applies an attention layer followed by an MLP (feed-forward) layer. What is the primary role of the MLP layer?

- ◯ It computes pairwise interactions between tokens to propagate information across the sequence.
- ◯ It enables parallel decoding by processing all tokens simultaneously.
- ● It applies a learned nonlinear transformation independently to each token's representation, expanding the model's capacity to represent complex features.
- ◯ It reduces the dimensionality of the hidden states to prevent overfitting in deeper layers.

Explanation: The attention layer mixes information across tokens, but applies relatively simple linear projections to each token. The MLP layer complements this by applying a nonlinear transformation to each token independently (the same function at every position), giving the model capacity to compute complex per-token features.

(c) **[1 Pt]** What is the primary benefit of using multi-head attention rather than single-head attention with the same total dimensionality?

- ◯ Each head operates on a distinct section of the input sequence, enabling the model to process longer contexts.
- ◯ The heads share weights with each other, reducing the total parameter count compared to single-head attention.
- ◯ Multiple heads enable parallel token generation during autoregressive decoding.
- ● Each head can learn a different attention pattern allowing the model to capture diverse types of dependencies simultaneously.

Explanation: Each head independently computes queries, keys, and values in its own low-dimensional subspace, so different heads can specialize in different types of relationships within the same sequence. This is analogous to the multiple channels in convolutional layers.

(d) **[1 Pt]** During next-token prediction training, how many **forward-passes** does a transformer have to do to compute the loss function for a sequence of length N? Assume we can use an arbitrarily large batch size.

- ● $\mathcal{O}(1)$, since the entire sequence can be processed in parallel.
- ○ $\mathcal{O}(N)$, because the model can process one element at a time.
- ○ $\mathcal{O}(N^2)$, because of the attention layer's quadratic complexity that is computed once for the whole sequence.
- ○ $\mathcal{O}(N^3)$, because of the attention layer's quadratic complexity that must be recomputed for every processed element of the sequence.

Explanation: Training can occur in parallel there is no dependency on previous model outputs–the target is predefined in the training dataset.

(e) **[1 Pt]** Consider the standard self-attention operation in matrix form

$$\text{Attn}(\mathbf{X}) = \text{softmax}\left(\frac{\mathbf{X}\mathbf{W}_Q\mathbf{W}_K^\top\mathbf{X}^\top}{\sqrt{d_k}}\right)\mathbf{X}\mathbf{W}_V,$$

where $\mathbf{X} \in \mathbb{R}^{N\times d}$ is the matrix of token embeddings (without any positional encoding). Which of the following is true about this operation?

*Note: We use attention **scores** to refer to Attn$(\mathbf{X}) \in \mathbb{R}^{N\times d}$ and attention **weights** to refer to the input to the softmax function in the equation above.*

- ◯ Reordering the rows of X does not change the attention *scores*.
- ● Reordering the rows of X reorders the rows of the attention *scores* in the same way.
- ◯ Reordering the rows of X changes the attention *weights* but not the attention *scores*.
- ◯ Reordering the rows of X has an unpredictable effect that depends on the learned weights $\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V$.

Explanation: Self-attention is permutation-equivariant. Each output row depends on the pairwise similarities between that row's query and all other rows' keys, followed by a row-wise softmax and weighted sum of values. Shuffling the input rows shuffles which queries, keys, and values land where, but the same pairwise computations still occur, so the output rows are shuffled in the same way. This is why positional encoding is needed: without it, the network cannot distinguish different orderings of the same tokens.

(f) **[6 pts] Attention Biases**

The full self-attention layer, as originally implemented, would calculate the query of input $\mathbf{x}_i$, and the key of input $\mathbf{x}_j$, for $j = 1, \ldots, N$, as

$$\mathbf{q}_i = \mathbf{W}_Q^\top\mathbf{x}_i + \mathbf{b}_Q \quad \text{and} \quad \mathbf{k}_j = \mathbf{W}_K^\top\mathbf{x}_j + \mathbf{b}_K.$$

Then, the pre-attention scores would be computed as

$$s_{ij} = \frac{\mathbf{q}_i^\top\mathbf{k}_j}{\sqrt{d_k}},$$

where d_k is the dimension of $\mathbf{q}_i$ and $\mathbf{k}_j$, and the softmax would be applied to the vector of pre-attention scores between $\mathbf{q}_i$ and $\mathbf{k}_1, \mathbf{k}_2, \ldots, \mathbf{k}_N$.

i. **[2 pts]** Let $\mathbf{x}$ be a vector and Softmax be the vector-softmax operation. Prove that, for any scalar c

$$\text{Softmax}(\mathbf{x} + c\mathbf{1}) = \text{Softmax}(\mathbf{x})$$

Solution: Let $y = \text{Softmax}(\mathbf{x} + c\mathbf{1})$, and y_i be its i^{th} component. Then

$$y_i = \frac{e^{x_i+c}}{\sum_j e^{x_j+c}} = \frac{e^c e^{x_i}}{\sum_j e^c e^{x_j}} = \frac{e^c e^{x_i}}{e^c \sum_j e^{x_j}} = \frac{e^{x_i}}{\sum_j e^{x_j}} = z_i$$

where $z = \text{Softmax}(\mathbf{x})$, which completes the proof.

ii. **[4 pts]** Show that the bias term b_K does not affect the final attention score and, thus, can be safely removed.

Solution: For any j, we have

$$\mathbf{q}_i^\top \mathbf{k}_j = (\mathbf{W}_Q^\top \mathbf{x}_i + \mathbf{b}_Q)^\top (\mathbf{W}_K^\top \mathbf{x}_j + \mathbf{b}_K) = (\mathbf{x}_i^\top \mathbf{W}_Q + \mathbf{b}_Q^\top)(\mathbf{W}_K^\top \mathbf{x}_j + \mathbf{b}_K)$$

which can be expanded out as

$$\mathbf{x}_i^\top \mathbf{W}_Q \mathbf{W}_K^\top \mathbf{x}_j + \mathbf{b}_Q^\top \mathbf{W}_K^\top \mathbf{x}_j + x_i^\top \mathbf{W}_Q \mathbf{b}_k + \mathbf{b}_Q^\top \mathbf{b}_K.$$

The term $\mathbf{b}_Q^\top \mathbf{b}_K$ is the same for all $j = 1, \ldots, N$, and therefore (using the previous result) id does not affect the softmax scores.
The term $\mathbf{x}_i^\top \mathbf{W}_Q \mathbf{b}_k$ depends on $\mathbf{x}_i$, but it is the same for all $j = 1, \ldots, N$. Therefore, it does not matter either.
Following from the previous part, the remaining relevant terms are

$$\mathbf{x}_i^\top \mathbf{W}_Q \mathbf{W}_K^\top \mathbf{x}_j + \mathbf{b}_Q^\top \mathbf{W}_K^\top \mathbf{x}_j = (\mathbf{W}_Q^\top \mathbf{x}_i + \mathbf{b}_Q)^\top \mathbf{W}_K^\top \mathbf{x}_j$$

which does not depend on $\mathbf{b}_K$. Therefore, this bias term can be removed.

(g) **[5 pts] Attention Masks**

We are training a transformer-based language model for a specialized application where it is useful to use bidirectional attention for the final M tokens of the generation. For simplicity, assume that every generated sequence has a total length of $N + M$. The first N tokens are generated using standard autoregressive causal attention. The final M tokens are generated in parallel such that they can attend to all prior tokens (1 to N), as well as all other tokens within the final M block bidirectionally. Since the generation of the final M tokens is parallel, not autoregressive, the input token at the final $M - 1$ positions is a fixed special token.

For a standard attention mask matrix, let 1 denote that a query token at row i is allowed to attend to a key token at column j, and 0 denote that the token is masked from attention.

i. **[3 pts]** Write out the full attention mask matrix for $N = 4$ and $M = 2$.

Solution:

For $N = 4, M = 2$, the matrix is 6×6. The first 4 rows are lower triangular (causal). The last 2 rows are all 1s because the M tokens attend to all N tokens and all M tokens bidirectionally.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

ii. **[2 pts] Inference Complexity** Focus purely on the inference process for generating this sequence from scratch (assume no prefilled prompt; the model must generate the N tokens step-by-step, and then generate the M tokens).

How many **distinct** matrix multiplications are required to compute the key vectors for all tokens in the sequence during a single inference pass?

○ N	○ $N + M$	○ $N \cdot M$
● $N + 1$	○ $2N + M$	○ $(N + M)^2$

Explanation: The first N tokens are generated autoregressively, so each requires its own matrix-vector multiplication to produce a key vector, giving N multiplications. The final M tokens are bidirectional and can be processed simultaneously in a single batched matrix multiplication (projecting an $M \times d$ matrix by W_K). This gives $N + 1$ total.

You are done with the final! Congratulations!

Use this page to draw something tuff 🤭