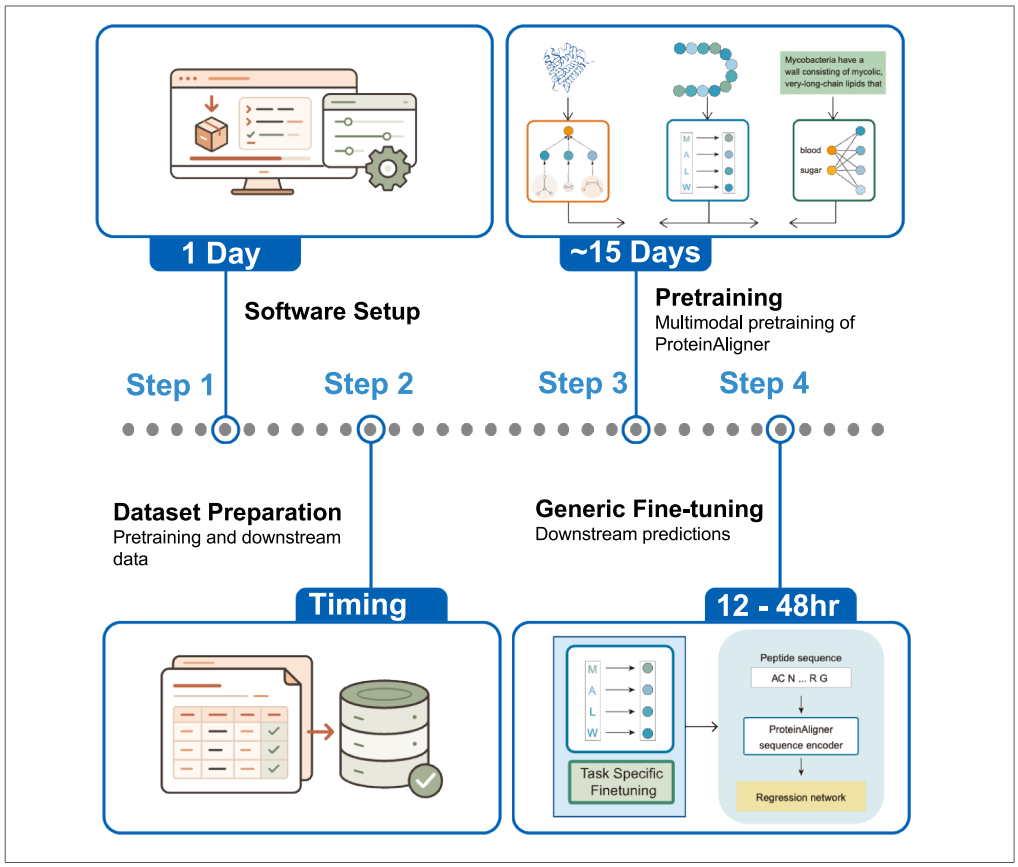
STAR Protocols

CellPress
OPEN ACCESS

Protocol

Protocol for tri-modal pretraining and downstream fine-tuning of protein representations using ProteinAligner



ProteinAligner integrates protein sequence, three-dimensional structure, and functional annotations in a shared embedding space. Here, we present a protocol for tri-modal pretraining and downstream fine-tuning of protein representations using ProteinAligner. We describe steps for preparing both paired sequence-structure and sequence-text datasets from public protein resources and pretraining ProteinAligner with distributed multimodal learning. We then detail procedures for fine-tuning downstream tasks and evaluating performance.

Publisher's note: Undertaking any experimental protocol requires adherence to local institutional guidelines for laboratory safety and ethics.

Han Guo, Li Zhang, Leah Schaffer, ..., Trey Ideker, Eric Xing, Pengtao Xie

p1xie@ucsd.edu

Highlights

Prepare paired sequence-structure and sequence-text pretraining datasets

Pretrain ProteinAligner by aligning sequence, structure, and text representations

Fine-tune pretrained protein representations across diverse downstream tasks

Verify training and prediction outputs with quantitative reference values

Guo et al., STAR Protocols 7, 104862
December 18, 2026 © 2026 The Authors. Published by Elsevier Inc.
https://doi.org/10.1016/j.xpro.2026.104862

STAR Protocols

CellPress
OPEN ACCESS

Protocol

Protocol for tri-modal pretraining and downstream fine-tuning of protein representations using ProteinAligner

Han Guo,[1,11] Li Zhang,[1,11] Leah Schaffer,[2] Young Su Ko,[3] Digvijay Singh,[4] Hamid Rahmani,[5] Danielle Grotjahn,[5] Elizabeth Villa,[4,6] Michael Gilson,[7] Wei Wang,[3,8] Trey Ideker,[2,9] Eric Xing,[10] and Pengtao Xie[1,2,12,13,*]

[1]Department of Electrical and Computer Engineering, University of California San Diego, La Jolla, CA 92093, USA
[2]Department of Medicine, University of California San Diego, La Jolla, CA 92093, USA
[3]Department of Chemistry and Biochemistry, University of California San Diego, La Jolla, CA 92093, USA
[4]School of Biological Sciences, University of California San Diego, La Jolla, CA 92093, USA
[5]Department of Integrative Structural and Computational Biology, The Scripps Research Institute, La Jolla, CA 92037, USA
[6]Howard Hughes Medical Institute, University of California San Diego, La Jolla, CA 92093, USA
[7]Skaggs School of Pharmacy and Pharmaceutical Sciences, University of California San Diego, La Jolla, CA 92093, USA
[8]Department of Cellular and Molecular Medicine, University of California San Diego, La Jolla, CA 92093, USA
[9]Department of Bioengineering, University of California San Diego, La Jolla, CA 92093, USA
[10]Machine Learning Department, Carnegie Mellon University, Pittsburgh, PA 15213, USA
[11]These authors contributed equally
[12]Lead contact
[13]Technical contact
*Correspondence: p1xie@ucsd.edu
https://doi.org/10.1016/j.xpro.2026.104862

SUMMARY

ProteinAligner integrates protein sequence, three-dimensional structure, and functional annotations in a shared embedding space. Here, we present a protocol for tri-modal pretraining and downstream fine-tuning of protein representations using ProteinAligner. We describe steps for preparing both paired sequence-structure and sequence-text datasets from public protein resources and pretraining ProteinAligner with distributed multimodal learning. We then detail procedures for fine-tuning downstream tasks and evaluating performance.

For complete details on the use and execution of this protocol, please refer to Zhang et al.[1]

BEFORE YOU BEGIN

Protein representation learning is critical for understanding protein function, molecular mechanism, and biological activity, with broad applications in functional annotation, protein engineering, and therapeutic discovery.[2,3] More broadly, recent advances in computational biology have demonstrated the value of integrating heterogeneous biological information and developing representations tailored to complex molecular and genomic relationships. Representative examples include multisource and multiview learning for drug–target and drug–drug interaction prediction,[4,5] graph-based and isoform-level methods for single-cell RNA-sequencing analysis,[6,7] and attention-based modeling of nanopore signals for DNA methylation detection.[8] Systematic computational analyses and curated databases have also facilitated genomic-island identification and the characterization of disease-associated viral mutations.[9,10] Collectively, these developments highlight the broader importance of combining complementary biological information through advanced computational representations. Numerous computational methods have been developed to learn protein representations from amino acid sequences, and

This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

large-scale protein language models have achieved strong performance across diverse downstream tasks. However, sequence-only representations do not fully capture the complementary information available for proteins. Three-dimensional structures provide physically grounded spatial context, while text descriptions contain experimentally curated functional and biological annotations that may not be directly inferable from sequence alone.[11–13] To address this limitation, we developed ProteinAligner, a multimodal protein representation-learning framework that aligns protein sequence, structure, and text information within a unified training workflow. In the ProteinAligner pipeline, protein sequence–text pairs and protein sequence–structure pairs are first collected and preprocessed from public protein resources. Protein representations are then trained to align complementary modalities, allowing the model to integrate evolutionary, structural, and functional information during pretraining. The resulting representations can subsequently be adapted to downstream protein prediction tasks through task-specific fine-tuning.

In this protocol, we present a detailed step-by-step instruction for executing the ProteinAligner workflow. We describe how to prepare the pretraining and downstream datasets, configure the software environment, launch multimodal pretraining, perform downstream fine-tuning, and verify the expected outputs. This protocol is intended to help researchers reproduce the ProteinAligner training and evaluation workflow and adapt it to related multimodal protein representation-learning tasks.

Innovation

Existing protein representation-learning techniques predominantly utilize amino acid sequence information, and recent protein foundation models have achieved strong performance by pretraining transformer architectures on large-scale sequence corpora.[2,3] However, sequenceonly pretraining does not explicitly incorporate complementary information from protein three-dimensional structures or functional text descriptions. ProteinAligner addresses this limitation by using the amino acid sequence as an anchor modality to align structural and textual protein information within a unified representation space. Particularly, ProteinAligner uses ESM-2 to encode protein sequences and ESM-IF1 to encode protein structures, while a vanilla 8-layer Transformer encoder is used to encode text descriptions derived from experimentally supported protein annotations.

The sequence encoder generates residue-level sequence representations with a shape of $L_{seq} \times 1280$, where L_{seq} is the length of the amino acid sequence. The structure encoder generates residue-level structure representations with a shape of $L_{str} \times 512$, where L_{str} is the number of residues represented in the input protein structure. The text encoder generates token-level text representations with a shape of $L_{text} \times 1024$, where L_{text} is the number of text tokens. These representations are then projected into a shared 1024-dimensional embedding space.

Software and project setup

Timing: 0.5–1 day

Note: ProteinAligner pretraining was developed on Linux x86_64 with Python 3.10 and PyTorch 2.1.0 built against CUDA 12.1. The Conda environment build and the released mapping archive account for most of the setup time. This section establishes the environment and project layout required for the pretraining workflow.

1. Verify the host requirements and retrieve the ProteinAligner repository.
   a. Confirm that the host uses Linux x86_64.
   b. Reserve several hundred GB of free disk space for the FASTA and PDB caches and rolling checkpoint history and at least 64 GB RAM on the head node.

c. Clone the public repository:
```
git clone https://github.com/Alexiland/ProteinAligner proteinaligner_star
```
d. Check out the versioned ProteinAligner commit:
```
git -C proteinaligner_star checkout 3bb238f8e9523121f8ccbe270e2bb0f90929528d
```
e. Change to the pretraining directory:
```
cd proteinaligner_star/pretrain
```

Note: The repository keeps pretraining code under pretrain/ and downstream-task code under ProteinAligner_downstream/. The ESM code needed by the project is bundled under pretrain/esm/; no separate ESM checkout is required for the baseline workflow.

2. Create a Conda virtual environment for pretraining.
   a. Run the following command:
   ```
   conda create -n proteinaligner_pretrain python=3.10 -y
   ```
   b. Activate the environment named proteinaligner_pretrain:
   ```
   conda activate proteinaligner_pretrain
   ```

   Note: The environment name proteinaligner_pretrain is distinct from the downstream proteinaligner environment, which prevents the pretraining setup from altering the downstream environment.

3. Install GPU-aware PyTorch and the remaining dependencies in three passes.
   a. Strip the PyTorch family and torch-scatter from the project requirements:
   ```
   grep -vE '^(torch|torchvision|torchaudio|pytorch-triton|torch-scatter)==' requirements.txt > /tmp/req.txt
   ```
   b. Install PyTorch 2.1.0 with CUDA 12.1 wheels:
   ```
   pip install --index-url https://download.pytorch.org/whl/cu121 torch==2.1.0 torchvision==0.16.0 torchaudio==2.1.0
   ```
   c. Install the remaining requirements:
   ```
   pip install -r /tmp/req.txt
   ```
   d. Install torch-scatter from the matching PyG wheel index:
   ```
   pip install torch-scatter -f https://data.pyg.org/whl/torch-2.1.0+cu121.html
   ```

   Note: Installing the dependencies in this order prevents the CUDA 12.1 PyTorch wheel from being overwritten by the version pinned in requirements.txt.

   ⚠ CRITICAL: torch-scatter ships pre-built wheels keyed to a specific PyTorch and CUDA combination. A wheel index that does not match the PyTorch version installed in step 3b causes a runtime import failure.

4. Prepare and document the project directory layout under pretrain/.
   a. Create the data, checkpoint, and script folders:
   ```
   mkdir -p pretrain/{data/fasta,data/pdb,data/mappings,ckpt,scripts}
   ```
   b. Unpack the released mapping archive into data/mappings/:
   ```
   unzip -o proteinaligner_pretrain_data.zip -d pretrain/data/mappings/
   ```
   c. Store the repository checkout, processed manifests, and saved checkpoints under versioned directory names.
   d. Record the exact Python, CUDA, and PyTorch versions, GPU model and count, and repository commit hash.

Note: Versioned directory names allow later downstream runs to be traced to a specific pretraining state.

⚠ CRITICAL: Complete environment and repository records are required to reproduce the pretraining run.

Dataset preparation for pretraining

Timing: 1–2 days, depending on download speed and preprocessing implementation

5. Prepare the pretraining sequence–text corpus from UniProtKB/Swiss-Prot.
   a. Download the UniProtKB/Swiss-Prot release used for the study and extract entries with function descriptions.
   b. Retrieve the protein sequence associated with each selected primaryAccession.
   c. Store each sequence and its paired text description in the format expected by the training code.
   d. Strip {ECO:NNNN} evidence tags and collapse line-wrap hyphens in each function description.
   e. Use the same canonical accession in the FASTA filename, FASTA header, and manifest uniprot_id field. Note: Each sequence is written as a single-record FASTA file at pretrain/data/fasta/<UNIPROT_ID>.fasta using the canonical Swiss-Prot header format, for example >sp|ACC|-NAME. The corresponding functional annotation is stored in pretrain/data/mappings/uniprot_seq_function_mapping_300.json as a JSON record with the keys uniprot_id and function.

```
cd pretrain/data

curl -O https://ftp.uniprot.org/pub/databases/uniprot/current_release/\

     knowledgebase/complete/uniprot_sprot.dat.gz

curl -O https://ftp.uniprot.org/pub/databases/uniprot/current_release/\

     knowledgebase/complete/uniprot_sprot.fasta.gz

gunzip -f uniprot_sprot.dat.gz uniprot_sprot.fasta.gz

python - <<'PY'

import json, os, re

from Bio import SeqIO, SwissProt

os.makedirs("fasta", exist_ok=True)

os.makedirs("mappings", exist_ok=True)

def clean_function(text):

    # Strip {ECO:...} evidence tags, collapse line-wrap hyphens,

    # drop trailing period

    text = re.sub(r"\s*\{ECO:[^\}]+\}", "", text)

    text = text.replace("- ", "-")

return text.rstrip(". ")

# Pass 1: parse .dat for entries with a FUNCTION annotation and len <= 300
```

```
kept = {}
with open("uniprot_sprot.dat") as fh:
    for rec in SwissProt.parse(fh):
      if len(rec.sequence) > 300:
        continue
      funcs = [c[len("FUNCTION: "):] for c in rec.comments
        if c.startswith("FUNCTION:")]
      if not funcs:
        continue
      kept[rec.accessions[0]] = clean_function(" ".join(funcs))
# Pass 2: split the canonical .fasta into per-accession files
for rec in SeqIO.parse("uniprot_sprot.fasta", "fasta"):
    acc = rec.id.split("|")[1] # rec.id = "sp|ACC|NAME"
    if acc not in kept:
      continue
    SeqIO.write([rec], open(f"fasta/{acc}.fasta", "w"), "fasta")
records = [{"uniprot_id": acc, "function": kept[acc]} for acc in kept]
json.dump(records, open("mappings/uniprot_seq_function_mapping_300.json", "w"))
print("seq-text records:", len(records))
PY
```

Note: The primaryAccession in the FASTA header is the canonical identifier used throughout the downstream workflow. The.dat release identifies entries with a FUNCTION: annotation and provides the function text. The per-entry FASTA shards originate from the.fasta release, which preserves the published OS/OX/GN/PE/SV header fields.

⚠ CRITICAL: Inconsistent accessions across the filename, FASTA header, and manifest prevent successful joins during training.

6. Prepare the pretraining sequence–structure corpus from RCSB PDB.
   a. Obtain candidate PDB identifiers from the PDB cross-references in the same reviewed UniProtKB/Swiss-Prot release used to prepare the FASTA files.
   b. Download the current RCSB/wwPDB PDB-format mirror.
   c. Convert each four-character PDB ID to lowercase and construct its archive path from characters 2 and 3.
   d. Compare every protein chain in each candidate PDB entry with the corresponding UniProt FASTA sequence.
   e. Extract the sequence-consistent source chain as a single-chain PDB file.
   f. Rename the extracted chain to A.
   g. Retain no more than one pair per UniProt accession and per PDB ID.
   h. Record the source-chain identifier, normalized chain A, and sequence-verification result in the manifest.

i. Exclude proteins longer than 300 residues before pretraining. Note: The 300-residue threshold addresses memory constraints during large-scale multimodal pretraining; it is not a biologically meaningful boundary or an intrinsic architectural limit of ProteinAligner. The residue-level inputs and self-attention layers make longer inputs substantially more expensive. This filter reduces padding and memory use and enables consistent batching of sequence–structure and sequence–text examples. After filtering, the pretraining corpus contained 133,726 sequence–structure pairs and 290,480 sequence–text pairs.

```
cd pretrain/data
mkdir -p pdb pdb_raw pdb_mirror mappings
rsync -rlpt -v -z --delete --port=33444 \
    rsync.rcsb.org::ftp_data/structures/divided/pdb/ pdb_mirror/
python - <<'PY'
import gzip, json, shutil
from pathlib import Path
from Bio import Align, SeqIO
from Bio.PDB import PDBIO, PDBParser
import esm.inverse_folding.util as ifu
selected = {r["uniprot_id"]: r["function"] for r in json.load( \
    open("mappings/uniprot_seq_function_mapping_300.json"))}
# Candidate IDs come from PDB cross-references in reviewed UniProtKB.
candidates = []
for rec in SeqIO.parse("uniprot_sprot.dat", "swiss"):
    uid = rec.accessions[0]
    if uid not in selected:
      continue
    for xref in rec.dbxrefs:
      if xref.startswith("PDB:"):
          candidates.append((uid, xref.split(":", 1)[1].upper()))
candidates = sorted(set(candidates))
aligner = Align.PairwiseAligner()
aligner.mode = "global"
aligner.match_score = 2
aligner.mismatch_score = -1
aligner.open_gap_score = -10
aligner.extend_gap_score = -0.5
def match_metrics(reference, query):
    """Identity over aligned residues and coverage of the UniProt sequence."""
    aln = aligner.align(reference, query)[0]
    pairs = []
```

```
    for (r0, r1), (q0, q1) in zip(aln.aligned[0], aln.aligned[1]):
    pairs.extend(zip(reference[r0:r1], query[q0:q1]))
  pairs = [(a, b) for a, b in pairs if a != "X" and b != "X"]
  if not pairs:
    return 0.0, 0.0
  identity = sum(a == b for a, b in pairs) / len(pairs)
  coverage = len(pairs) / len(reference)
  return identity, coverage
# Code continues in the following text box.
```

```
# Continued from the preceding text box.
parser = PDBParser(QUIET=True)
keep, rejected = [], []
used_uniprot, used_pdb = set(), set()
def reject(uid, pid, reason):
    rejected.append({"uniprot_id": uid, "pdb_id": pid,"reason": reason})
for uid, pid in candidates:
    if uid in used_uniprot or pid in used_pdb:
      continue
    if len(pid) != 4:
      reject(uid, pid, "not_available_as_a_four-character_PDB_ID")
      continue
    reference = str(next(SeqIO.parse(f"fasta/{uid}.fasta", "fasta")).seq).upper()
    if len(reference) > 300:
      continue
    pid_lower = pid.lower()
    middle_two = pid_lower[1:3]
    archive_file = Path("pdb_mirror") / middle_two / f"pdb{pid_lower}.ent.gz"
    if not archive_file.exists():
      reject(uid, pid, "PDB_format_archive_file_missing")
      continue
    ent_file = Path("pdb_raw") / f"pdb{pid_lower}.ent"
    if not ent_file.exists():
      with gzip.open(archive_file, "rb") as source, open(ent_file, "wb") as
    target:
      shutil.copyfileobj(source, target)
    # SEQRES records retain the deposited chain identifiers.
    chain_sequences = {
```

```
      rec.annotations["chain"]: str(rec.seq).upper()
      for rec in SeqIO.parse(ent_file, "pdb-seqres")
      if rec.annotations.get("chain")
    }
if not chain_sequences:
    reject(uid, pid, "no_protein_chain_sequence")
    continue
    scored = [(match_metrics(reference, seq), chain_id, seq)
      for chain_id, seq in chain_sequences.items()]
    scored.sort(key=lambda x: (-x[0][0], -x[0][1], x[1]))
    (identity, coverage), source_chain_id, source_sequence = scored[0]
# Code continues in the following text box.
```

```
# Continued from the preceding text box.
  conflicting_ties = [x for x in scored[1:]
    if x[0] == scored[0][0] and x[2] != source_sequence]
  if identity < 0.999999 or conflicting_ties:
    reject(uid, pid, "ambiguous_or_sequence_inconsistent_chain")
    continue
  structure = parser.get_structure(pid, ent_file)
  model = next(structure.get_models())
  if source_chain_id not in model:
    reject(uid, pid, "matched_chain_has_no_coordinates")
    continue
  source_chain = model[source_chain_id]
  for chain in list(model):
    if chain is not source_chain:
      model.detach_child(chain.id)
  source_chain.id = "A"
  pdb_path = Path("pdb") / f"{pid_lower}.pdb"
  writer = PDBIO()
  writer.set_structure(structure)
  writer.save(str(pdb_path))
  # Reload the normalized file exactly as the training Dataset does.
  try:
    _, normalized_sequence = ifu.load_coords(str(pdb_path), "A")
  except Exception as exc:
    pdb_path.unlink(missing_ok=True)
```

```
    reject(uid, pid, f"normalized_chain_load_failed:{type(exc).__name__}")
    continue
  output_identity, output_coverage = match_metrics(
    reference, normalized_sequence.upper())
  if output_identity < 0.999999:
    pdb_path.unlink(missing_ok=True)
    reject(uid, pid, "normalized_chain_sequence_mismatch")
    continue
  keep.append({"uniprot_id": uid, "pdb_id": pid,
    "function": selected[uid],
    "source_chain_id": source_chain_id,
    "normalized_chain_id": "A",
    "sequence_identity": round(output_identity, 6),
    "sequence_coverage": round(output_coverage, 6),
    "sequence_match": "verified"})
  used_uniprot.add(uid)
  used_pdb.add(pid)
json.dump(keep, open("mappings/uniprot_seq_str_mapping_300.json", "w"))
json.dump(rejected, open("mappings/uniprot_seq_str_rejections.json", "w"),
    indent=2)
print("seq-structure pairs:", len(keep))
print("rejected candidates:", len(rejected))
PY
```

Note: Wrap archive decompression and PDB parsing in exception handling; record gzip_decompression_failed or pdb_parse_failed in the rejection manifest and continue to the next candidate.

Note: The current RCSB/wwPDB PDB-format mirror is rsync.rcsb.org::ftp_data/structures/divided/pdb/ and uses rsync port 33444. Each archive filename contains the prefix pdb, the full lowercase PDB ID, and the suffix.ent.gz; characters 2 and 3 form its subdirectory. For example, PDB ID 4HHB is stored as hh/pdb4hhb.ent.gz. The normalized output is pdb/4hhb.pdb, and no coordinate-format conversion is required.

Note: Entries without a usable PDB-format file or a verified matched coordinate chain are recorded in mappings/uniprot_seq_str_rejections.json and excluded explicitly.

Note: Chain A in the training Dataset is the normalized identifier in the derived single-chain PDB file. It does not refer unconditionally to chain A of the original deposited structure. The original matched identifier remains available as source_chain_id in the sequence– structure manifest.

7. Assemble the paired dataset used by pretraining.

a. Build the sequence–text manifest under pretrain/data/mappings/using the schema defined in step 1c.
b. Build the sequence–structure manifest under pretrain/data/mappings/.
   i. Retain the UniProt accession, PDB ID, and function in each record.
   ii. Add the original and normalized chain identifiers.
   iii. Record the alignment identity, coverage, and verification status.

Note: In each sequence–structure record, the exact JSON keys are those shown in the code below.

c. Confirm that the UniProt and PDB identifiers are one-to-one.
d. Confirm that every normalized PDB contains chain A.
e. Confirm that every retained sequence–structure record has a verified sequence match.

```
cd pretrain

python - <<'PY'

import json, os

import esm.inverse_folding.util as ifu

D = "data"

ss = json.load(open(f"{D}/mappings/uniprot_seq_str_mapping_300.json"))

sf = json.load(open(f"{D}/mappings/uniprot_seq_function_mapping_300.json"))

required = {"uniprot_id", "pdb_id", "source_chain_id",

  "normalized_chain_id", "sequence_identity",

  "sequence_coverage", "sequence_match"}

assert all(required <= set(r) for r in ss)

assert len({r["uniprot_id"] for r in ss}) == len(ss)

assert len({r["pdb_id"] for r in ss}) == len(ss)

validated_ss = []

for r in ss:

  fasta_path = f"{D}/fasta/{r['uniprot_id']}.fasta"

  pdb_path = f"{D}/pdb/{r['pdb_id'].lower()}.pdb"

  if (not os.path.exists(fasta_path) or not os.path.exists(pdb_path)

    or r["normalized_chain_id"] != "A"

    or r["sequence_match"] != "verified"):

    continue

  _, pdb_sequence = ifu.load_coords(pdb_path, "A")

  if not pdb_sequence:

    continue

  validated_ss.append®

ss = validated_ss

sf = [r for r in sf
```

```
  if r.get("function")
  and os.path.exists(f"{D}/fasta/{r['uniprot_id']}.fasta")]
json.dump(ss, open(f"{D}/mappings/uniprot_seq_str_mapping_300.json", "w"))
json.dump(sf, open(f"{D}/mappings/uniprot_seq_function_mapping_300.json", "w"))
print("validated seq-structure pairs:", len(ss))
print("validated seq-text pairs:", len(sf))
PY
```

f. Record the final sequence–structure and sequence–text pair counts used for training. Note: The filtered corpus in the original manuscript contained 133,726 sequence– structure pairs and 290,480 sequence–text pairs.

Pause point: Processed pretraining manifests can be stored and reused before model training.

Dataset preparation for downstream tasks

Timing: 0.5–1 day, depending on download speed and preprocessing implementation

Note: Each task below is shown with the preprocessing code from its source repository. Absolute paths appear as placeholders (<...>), and <proteinaligner_ckpt> denotes the pretrained ProteinAligner checkpoint.

8. Obtain each task dataset from its public source repository.
   a. Clone the repository for the selected downstream task.
   b. Check out the exact commit reported in the key resources table:

```
git -C <repository> checkout <commit>
```

Note: The datasets are distributed within their respective source repositories; no separate dataset download is required.

Note: Pathogenic missense variants (VariPred). The VariPred repository contains ClinVarderived missense variants as train/test lists in HGVSp notation with binary benign/pathogenic labels: https://github.com/wlin16/VariPred.

Note: Thermostability (HotProtein). The HotProtein repository contains protein sequences paired with thermostability labels or values and the official benchmark split: https://github.com/VITA-Group/HotProtein.

Note: Type I anti-CRISPR activity detection (AcrTransAct). The AcrTransAct repository contains a curated table of Acr–CRISPR pairs with binary inhibition labels: https://github.com/USask-BINFO/AcrTransAct.

Note: Bioactive peptide prediction (UniDL4BioPep). The UniDL4BioPep repository contains 19 bioactivity datasets as paired train/test files of peptide sequences with binary activity labels: https://github.com/dzjxzyd/UniDL4BioPep.

Note: MIC prediction (Deep_AMP). The Deep_AMP repository contains antimicrobial peptide sequences paired with continuous minimum-inhibitory-concentration values: https://github.com/amirpandi/Deep_AMP.

9. Apply the official train–test split to VariPred, HotProtein, and UniDL4BioPep.

Note: Deep_AMP does not provide an official test split; its repository supplies the Gramnegative and Gram-positive MIC datasets only as training CSVs.

10. Create a fixed-seed split for each task without an official split.
    a. Create a stratified 80:20 split for Type I anti-CRISPR detection using random_state=41.
    b. Create an unstratified 80:20 split of the Gram-negative E. coli Deep_AMP data using random_state=42.
    c. Apply the same fixed split to every representation model and model-training seed.

⚠ CRITICAL: Changes in train/test membership would confound comparisons across representation models and model-training seeds.

11. Convert each dataset into the task-specific input format required by the downstream training code.
    a. Materialize the protein or peptide sequence or sequences for the variant-, peptide-, and AMP-level tasks.
    b. Load the pretrained ProteinAligner weights into the sequence encoder.
    c. Compute the ProteinAligner embedding for each sequence.
    d. Pair each Acr sequence with its CRISPR sequence before encoding the anti-CRISPR dataset.
12. Preprocess each task dataset.
    a. Prepare the pathogenic missense variant data (VariPred).
        i. Clone the VariPred repository.
        ii. Parse each HGVSp variant in the official split.
        iii. Resolve each wild-type sequence from the local cache or NCBI Entrez.
        iv. Build the mutant sequence and validate the residue match.

Note: The official split contains whitespace-separated tables with a target_id column in HGVSp notation and a binary label column.

```
git clone https://github.com/wlin16/VariPred

cd VariPred/Dataset_preparation

# Input: VariPred_train.txt / VariPred_test.txt

#    columns: target_id (e.g. NP_689699.2:p.Gly56Ser), label (0/1)

# Set a valid address in prepare_dataset.py (Entrez.email), then run:

python prepare_dataset.py VariPred_train

python prepare_dataset.py VariPred_test

# Output CSV columns:

# target_id, NP_id, aa_index, Length, wt_aa, mt_aa, wt_seq, mt_seq, label
```

    b. Prepare the thermostability data (HotProtein).
        i. Clone the HotProtein repository.
        ii. Load the released sequence and label files with the repository's data-loading utilities.

iii. Retain the official benchmark split.

```
git clone https://github.com/VITA-Group/HotProtein

# Load the released sequence/label files via the repository's data

# loaders; keep the benchmark's official split.
```

c. Prepare the Type I anti-CRISPR data (AcrTransAct).
   i. Clone the AcrTransAct repository.
   ii. Retain rows marked for use and remove invalid Acr sequences.
   iii. Create the stratified 80:20 split using random_state=41.

Note: This procedure reproduces the repository's split_data() function.

```
git clone https://github.com/USask-BINFO/AcrTransAct

import pandas as pd

from sklearn.model_selection import train_test_split

df = pd.read_excel("data/Acr_CRISPR_inhibition_type_I.xlsx")

df = df[df["use"] == 1]

# keep valid entries

df = df[df["Acr_seq"].str.len() > 1]

train_df, test_df = train_test_split(

  df, test_size=0.2, random_state=41,

stratify=custom_stratify(df), # joint: label x CRISPR-system type

)

train_df.to_csv("train_val1.csv", index=False)

test_df.to_csv("test1.csv", index=False)
```

d. Prepare the bioactive peptide data (UniDL4BioPep).
   i. Clone the UniDL4BioPep repository.
   ii. Load the <task>_train.xlsx and <task>_test.xlsx files for each of the 19 tasks.
   iii. Load the ProteinAligner weights into the ESM backbone.
   iv. Generate one embedding for each sequence.

Note: Each task file contains sequence and label columns.

```
# git clone https://github.com/dzjxzyd/UniDL4BioPep

import torch, esm, pandas as pd

from copy import deepcopy

model, alphabet = esm.pretrained.esm1b_t33_650M_UR50S()

ckpt = torch.load("<proteinaligner_ckpt>", map_location="cuda")
```

```
prefix = "modality_trunks.sequence.seq_encoder."
state = {k.replace(prefix, ""): deepcopy(v)
  for k, v in ckpt["model"].items()
if prefix in k and k.replace(prefix, "") in model.state_dict()}
model.load_state_dict(state, strict=False)
model = model.cuda().eval()
# For each task file, mean-pool the last-layer token embeddings per
# sequence and save <task>_{train,test}_..._dimension.csv
```

e. Prepare the MIC-prediction data (Deep_AMP).
   i. Clone the Deep_AMP repository. ii. Create the fixed 80:20 split using random_state=42. iii. Save the derived files under data/train_data/AMP/proteinaligner_split/using the seed-42 filenames shown below. iv. Load the ProteinAligner checkpoint into the ESM-2 650M backbone. v. Mean-pool each sequence embedding and save the resulting output CSV. vi. Preserve the repository's source CSV without modification.

Note: The repository provides regressor_train_AMP_gr_neg.csv as a single Gramnegative E. coli table with sequence and value columns; it does not provide an official test CSV.

⚠ CRITICAL: Overwriting the source CSV would remove the unmodified reference data needed to reproduce the derived split.

```
# git clone https://github.com/amirpandi/Deep_AMP
from pathlib import Path
import torch, esm, collections, pandas as pd
from copy import deepcopy
from tqdm import tqdm
# 1. Load ESM-2 650M and overwrite it with the ProteinAligner weights
model, alphabet = esm.pretrained.esm2_t33_650M_UR50D()
batch_converter = alphabet.get_batch_converter()
ckpt = torch.load("<proteinaligner_ckpt>", map_location="cuda")
prefix = "model."
state = {k.replace(prefix, ""): deepcopy(v)
  for k, v in ckpt.items()
  if prefix in k and k.replace(prefix, "") in model.state_dict()}
model.load_state_dict(state, strict=False)
model = model.cuda().eval()
# 2. Per-sequence embedding: mean-pool the last-layer (33) tokens
def esm_embeddings(seq_list):
  _, _, tokens = batch_converter(seq_list)
```

```
  lens = (tokens != alphabet.padding_idx).sum(1)
  with torch.no_grad():
    out = model(tokens.cuda(), repr_layers=[33])["representations"][33]
  reps = [out[i, 1:lens[i] - 1].mean(0).tolist() # drop BOS/EOS
    for i in range(len(lens))]
  return pd.DataFrame(reps)
# 3. Embed the two derived seed-42 split files and save the tables
split_dir = Path("data/train_data/AMP/proteinaligner_split")
output_dir = Path("esm_representation")
output_dir.mkdir(exist_ok=True)
for split_name in (
  "regressor_train_AMP_gr_neg_seed42",
  "regressor_test_AMP_gr_neg_seed42",
):
  dataset = pd.read_csv(split_dir / f"{split_name}.csv")
  embeddings = pd.DataFrame()
  for seq in tqdm(dataset["sequence"]):
    one = esm_embeddings([(seq, seq)]) # ESM expects (name, seq)
    embeddings = pd.concat([embeddings, one])
embeddings.to_csv(output_dir / f"{split_name}.csv", index=False)
```

Pause point: Processed task-specific splits can be stored and reused before downstream model training.

13. Place the pretrained ESM-2 and ESM-IF1 weights in the local cache or checkpoint directory used by the codebase.

```
mkdir -p ~/.cache/torch/hub/checkpoints
cd ~/.cache/torch/hub/checkpoints
# Sequence encoder: ESM-2 (650M)
wget https://dl.fbaipublicfiles.com/fair-esm/models/\
esm2_t33_650M_UR50D.pt
wget https://dl.fbaipublicfiles.com/fair-esm/regression/\
esm2_t33_650M_UR50D-contact-regression.pt
# Structure encoder: ESM-IF1 (GVP-Transformer, 142M)
wget https://dl.fbaipublicfiles.com/fair-esm/models/\
esm_if1_gvp4_t16_142M_UR50.pt
# Pretrained ProteinAligner checkpoint (load into the ESM-2 backbone
# for downstream embedding generation).
```

14. Create subdirectories for raw downloads, processed manifests, logs, and checkpoints.

```
mkdir -p data/raw data/processed checkpoints logs
```

15. Save each processed dataset as a manifest or table with stable identifiers and paths that match the expected input format of the training scripts.

KEY RESOURCES TABLE

REAGENT or RESOURCE	SOURCE	IDENTIFIER
Deposited data		
UniProtKB/Swiss-Prot reviewed protein sequence and annotation files	UniProt Consortium[14]	Swiss-Prot DAT; Swiss-Prot FASTA; RRID:SCR_002380
RCSB Protein Data Bank PDB-format structure archive	RCSB PDB[15]	https://www.rcsb.org/docs/programmatic-access/file-download-services; rsync.rcsb.org::ftp_data/structures/divided/pdb/(port 33444)
ProteinAligner sequence– text and sequence–structure mapping manifests	This paper	Zenodo v1.1; uniprot_seq_function_mapping_300.json; uniprot_seq_str_mapping_300.json
ProteinAligner pretrained checkpoint and SHA-256 checksum	This paper	Zenodo v1.1 (https://doi.org/10.5281/zenodo.21891297); checkpoint-22.pth; checkpoint-22.pth.sha256
VariPred ClinVar-derived missense-variant train/test data	Lin et al.[16]	GitHub commit 49f66fb
HotProtein thermostability benchmark data and official split	Chen et al.[17]	GitHub commit 9cf0cba
AcrTransAct Type I Acr– CRISPR inhibition data	Hasani et al.[18]	GitHub commit a1fbf27
UniDL4BioPep 19 bioactive-peptide train/test datasets	Du et al.[19]	GitHub commit 23d025f
Deep_AMP Gram-negative and Gram-positive MIC datasets	Pandi et al.[20]	GitHub commit a44465f
Software and algorithms		
ProteinAligner v1.1 source code	This paper	GitHub commit 3bb238f8; Zenodo v1.1
ProteinAligner pinned Python dependency set	This paper	requirements.txt at commit 3bb238f8
Python 3.10	Python Software Foundation	Python 3.10 release
Conda environment and package manager	Anaconda, Inc.	https://docs.conda.io/projects/conda/
pip 23.2.1	Python Packaging Authority	PyPI: pip 23.2.1
Git version-control software	Software Freedom Conservancy	https://git-scm.com/
cURL, GNU Wget, rsync, and Info-ZIP unzip command-line utilities	cURL project; GNU Project; rsync project; Info-ZIP	https://curl.se/; https://www.gnu.org/software/wget/; https://rsync.samba.org/; https://infozip.sourceforge.net/
PyTorch 2.1.0, torchvision 0.16.0, and torchaudio 2.1.0	PyTorch Foundation	PyTorch 2.1.0 software stack; RRID:SCR_018536
NVIDIA CUDA Toolkit 12.1	NVIDIA	CUDA 12.1 download archive
fair-esm 2.0.1	Meta AI Research	facebookresearch/esm; vendored in ProteinAligner commit 3bb238f8
ESM-1b sequence model (esm1b_t33_650M_UR50S)	Meta AI Research[3]	model checkpoint
ESM-2 650M sequence model (esm2_t33_650M_ UR50D) and contactregression weights	Meta AI Research[2]	model checkpoint; contact-regression checkpoint
ESM-IF1 structure model (esm_if1_gvp4_t16_142M_ UR50)	Meta AI Research[21]	model checkpoint
ProteinAligner 8-layer Transformer text encoder	This paper	Included in ProteinAligner commit 3bb238f8

(*Continued on next page*)

Continued

REAGENT or RESOURCE	SOURCE	IDENTIFIER
Biopython 1.84	Biopython Project	PyPI: biopython 1.84;RRID:SCR_007173
NumPy 1.25.2	NumPy Developers	PyPI: numpy 1.25.2; RRID:SCR_008633
NumPy 1.18.1 (Deep_AMP environment)	NumPy Developers	PyPI: numpy 1.18.1; RRID:SCR_008633
pandas 1.0.2 (Deep_AMP environment)	pandas development team	PyPI: pandas 1.0.2; RRID:SCR_018214
TensorFlow 2.11.0 (Deep_AMP environment)	Google Brain team	PyPI: tensorflow 2.11.0; RRID:SCR_016345
scikit-learn 1.3.0	scikit-learn developers	PyPI: scikit-learn 1.3.0; RRID:SCR_002577
SciPy 1.11.2	SciPy developers	PyPI: scipy 1.11.2; RRID:SCR_008058
tqdm 4.66.1	tqdm developers	PyPI: tqdm 4.66.1
torch-scatter 2.1.1	PyTorch Geometric team	PyPI: torch-scatter 2.1.1; https://data.pyg.org/whl/torch-2.1.0+cu121.html
VariPred preprocessing and prediction code	Lin et al.[16]	GitHub commit 49f66fb
HotProtein data-loading and thermostabilityprediction code	Chen et al.[17]	GitHub commit 9cf0cba
AcrTransAct preprocessing and classification code	Hasani et al.[18]	GitHub commit a1fbf27
UniDL4BioPep embedding and classification code	Du et al.[19]	GitHub commit 23d025f
Deep_AMP MIC-regression code	Pandi et al.[20]	GitHub commit a44465f
ProtST comparison model	DeepGraphLearning[13]	https://github.com/DeepGraphLearning/ProtST
ESM-S comparison model	DeepGraphLearning	https://github.com/DeepGraphLearning/esm-s; https://huggingface.co/Oxer11/ESM-S
ESM3 1.4B comparison model (esm3-sm-open-v1)	EvolutionaryScale/Biohub	https://huggingface.co/biohub/esm3-sm-open-v1; https://github.com/evolutionaryscale/esm
ProTrek comparison model	Westlake-REPL	https://github.com/westlake-repl/ProTrek; https://huggingface.co/westlake-repl/ProTrek_650M
Other		
NVIDIA A100 GPU with 80GB memory	NVIDIA	NVIDIA A100 Tensor Core GPU
Five-node Linux x86_64 compute cluster with 40 A100 GPUs and a highbandwidth interconnect	Local institutional high-performance computing resource	N/A

Note: The UniProtKB and RCSB links above provide the current source archives for rebuilding the corpus. The ProteinAligner v1.1 Zenodo record preserves the versioned code, pretraining mapping manifests, pretrained checkpoint, and SHA-256 checksum used by this protocol.

STEP-BY-STEP METHOD DETAILS

This section separates the workflow into two major phases: multimodal pretraining and generic downstream fine-tuning.

Multimodal pretraining of ProteinAligner

Timing: ~15 days on 40 NVIDIA A100 GPUs

This stage aligns sequence, structure, and text representations with contrastive learning so that the resulting encoders can be reused across downstream tasks. The training writes rolling checkpoints and the training log under pretrain/ckpt/.

Note: The 20-epoch pretraining schedule was run on 40 NVIDIA A100 80 GB GPUs distributed across five nodes (8 GPUs per node) connected by a high-bandwidth interconnect and completed in approximately 15 days of wall-clock time. Throughput scales close to linearly with the number of GPUs at this batch size. The single-node, 2-GPU configuration shown

below is suitable for a smoke test or short sanity run but is not practical for the full 20-epoch schedule.

Note: Step 2 launches train_joint_encoder.py. The description below maps the published hyperparameters to that script's code paths. Readers who only need to reproduce the pretraining can proceed directly to steps 1 and 2.

Note: The sequence encoder is ESM-2 (esm2_t33_650M_UR50D); the structure encoder is the ESM-IF1 encoder (esm_if1_gvp4_t16_142M_UR50); the text encoder is a SimpleTransformer trained from scratch. The beginning-of-sequence token of ESM-2, the index-0 token of the ESM-IF1 encoder output, and the end-of-sequence token of the text encoder are used as the per-modality summaries. Each summary passes through LayerNorm →linear projection to the shared embedding dimension →L2-normalize →logit scaling. The contact-prediction and language-modeling heads of ESM-2 are discarded before training.

Note: Each iteration samples one batch from each manifest. Use the prepared sequence–structure manifest for the first batch and the prepared sequence–text manifest for the second. The two InfoNCE losses are added with equal weight. Equal weighting was adopted as a simple and interpretable default that avoids introducing an additional loss-weighting hyperparameter. Alternative fixed or adaptive weighting strategies were not systematically evaluated; therefore, this setting should not be interpreted as an empirically established optimum across modalities. The effective global batch size is –batch_size × –world_size × –accum_iter.

Note: The training script uses AdamW with the learning rate, weight decay, and temperature set on the command line. The launch command in step 2 uses lr = 5×10^{-6}, weight decay 0.05, and τ = 0.1, betas (0.9, 0.95), gradient clipping at max norm 1.0, mixed precision via NativeScaler, and CosineAnnealingLR with T_{max} = –epochs and η_{min} = lr/50.

Note: The following pseudocode illustrates one training step.

```
# z_*: L2-normalized projected modality summaries (shape [B, D])
# tau: --temperature
  for epoch in 1..E: # E = --epochs (20)
  for (seq_a, struct), (seq_b, text) in zip(loader_ss, loader_st):
  z_s_a = norm_proj(ESM2(seq_a)[<cls>] ) # sequence
  z_t = norm_proj( ESM_IF1(struct)[0]) # structure
  z_s_b = norm_proj(ESM2(seq_b)[<cls>] ) # sequence
  z_x = norm_proj( TextEnc(text)[<eos>] ) # text
  L_ss = InfoNCE(z_s_a, z_t, tau) # sequence <-> structure
  L_st = InfoNCE(z_s_b, z_x, tau) # sequence <-> text
  loss = L_ss + L_st # equal weight
  loss.backward()
  clip_grad_norm_(model.parameters(), max_norm=1.0)
  if (step % accum_iter) == 0:
optimizer.step(); scheduler.step(); optimizer.zero_grad()
# InfoNCE(u, v, tau), with u and v shaped [B, D]:
```

```
# z = cat([u, v], dim=0) # [2B, D]

# sim = cosine_similarity(z[:, None, :],

#  z[None, :, :], dim=-1) / tau

# self_mask = eye(2 * B, dtype=bool, device=z.device)

# sim.masked_fill_(self_mask, -inf)

# mask only exact self-pairs

# pos_mask = self_mask.roll(shifts=B, dims=0)

# paired cross-modal rows

# return (-sim[pos_mask] + logsumexp(sim, dim=-1)).mean()
```

Note: The mean is taken over all 2B anchors, so each modality pair contributes in both directions: sequence-to-structure and structure-to-sequence for Lss, and sequence-to-text and text-to-sequence for Lst. The two bidirectional losses are then added with equal weight.

Note: An extended pretraining objective incorporating direct structure–text alignment was also evaluated alongside the primary sequence–structure and sequence–text objectives. Structure and text may nevertheless become indirectly related through their respective alignment with the shared sequence space. The primary protocol retains sequence as the anchor because it accommodates incomplete modality coverage. The other proteins in each batch serve as negatives; the implementation does not use an external memory bank, feature queue, or explicit hard-negative-mining procedure. The effective batch size and uniformly sampled pairs therefore determine the number and diversity of negatives.

Note: The script logs three losses at every iteration: dual_nll for sequence–structure, dual_nll_2 for sequence–text, and their combined loss. It also records the learning rate in./ckpt/train.log. Rolling encoder checkpoints are written under./ckpt/ at the end of each epoch and contain the weights for the downstream-task code in ProteinAligner_downstream/.

Note: Complete steps 1 and 2 after preparing the datasets to launch pretraining.

1. Verify that the intended GPUs are visible to PyTorch before submitting the run.
   a. Print the visible device count and CUDA availability:
      `python -c "import torch; print(torch.cuda.device_count(), torch.cuda.is_available())"`
   b. Confirm that the command reports True and the expected number of visible GPUs
2. Launch distributed training from inside pretrain/.
   a. Change to the pretraining directory.
      `cd pretrain`
   b. Run the single-node, 2-GPU baseline:

```
NCCL_P2P_DISABLE=1 NCCL_IB_DISABLE=1 \

torchrun \

  --nnodes=1 --nproc-per-node=2 \

  --rdzv-backend=c10d --rdzv-endpoint=localhost:23308 --rdzv_id 234 \

  train_joint_encoder.py \

  --output_dir ./ckpt \
```

```
  --log_dir ./ckpt \
  --world_size 2 --batch_size 2 --accum_iter 1 \
  --epochs 20 --lr 5e-6 --weight_decay 0.05 --temperature 0.1 \
  --num_workers 4 --pin_mem \
2>&1 | tee ./ckpt/train.log
```

c. Set -nproc-per-node and -world_size to match the available GPUs.
d. Add -nnodes and a shared -rdzv-endpoint when using more than one node.
e. Monitor ./ckpt/train.log for the separate sequence–structure, sequence–text, and combined losses.
f. Retain the rolling checkpoints written under ./ckpt/ at the end of each epoch.

Note: The published configuration used 40 GPUs across five A100 nodes. Set –nnodes to 5, –nproc-per-node to 8, and –world_size to 40. Point –rdzv-endpoint to the rank-0 host. Keep –batch_size and –accum_iter as shown to match the published effective global batch size. If memory permits, increase –batch_size and decrease –accum_iter proportionally.

⚠ CRITICAL: The single-node smoke-test command disables peer-to-peer and InfiniBand transports with NCCL_P2P_DISABLE=1 and NCCL_IB_DISABLE=1. Unset both variables on a multinode A100 cluster with working InfiniBand to recover full collective bandwidth.

⚠ CRITICAL: Pretraining is the longest-running stage of the protocol. Confirm the available compute time before launching the full schedule.

Note: The versioned ProteinAligner v1.1 release, including checkpoint-22.pth and its SHA-256 checksum, is archived in Zenodo v1.1.

Generic fine-tuning workflow for downstream prediction

Timing: 12–24 h per task

This major step reuses a pretrained encoder for a downstream task and pairs it with a taskspecific prediction head.

Note: In this generic overview, angle-bracket tokens denote task-specific user-supplied paths or identifiers. The complete Deep_AMP worked example below uses literal split and output paths and defines its single user-specific checkpoint-path token explicitly.

3. Select the downstream task and choose the corresponding pretrained encoder.
   a. Use the sequence encoder for pathogenic missense variant prediction, type I anti- CRISPR activity detection, peptide bioactivity prediction, and MIC prediction.
   b. Use the structure encoder for thermostability prediction.
4. Load the processed train–test split and adapt the task head to the target label space.
   a. Clone the task repository and change to its root directory.
   b. Install the repository's pinned requirements.
   c. Place the ProteinAligner checkpoint where the embedding script loads the encoder backbone.
   d. Load the processed train–test split.
   e. For classification tasks, define the final layer to match the number of classes.

f. For MIC prediction, define a regression head and the corresponding loss function.

```
# Common setup (run once per task repository)
git clone <task_source_repository>
cd <task_repository>
pip install -r requirements.txt
# Place the pretrained ProteinAligner checkpoint where the repo's
# embedding script loads the backbone, and load its weights into the
# ESM-2 650M encoder (strip the "model." prefix from the state dict).
# All subsequent embeddings are then ProteinAligner embeddings.
# Pathogenic missense variants (VariPred): the binary label space is
# read from the 'label' column; no separate head config is required.
# Type I anti-CRISPR (AcrTransAct): choose the classification head and
# the input feature mode (1 = sequence embeddings).
python3 code/training.py --model_type=CNN --feature_mode=1 \
  --esm_version=650m --run_mode=train
# Peptide bioactivity (UniDL4BioPep): in the task notebook's ESM_CNN
# cell, set the input embedding dimension (e.g. 1280) and the number
# of classes (2 for each binary bioactivity task).
# MIC (Deep_AMP): select the encoder backbone; the regression head and
# MSE loss are the defaults.
python train_regressor.py --model_architecture esm --data_type esm
```

5. Fine-tune the prediction head with the task-specific optimizer and stopping criteria.
   a. Use a multilayer perceptron head, Adam, and early stopping for pathogenic missense variant prediction.
   b. Use a multilayer perceptron head on the structure embeddings for thermostability prediction.
   c. Use convolutional classification heads for Type I anti-CRISPR activity detection and peptide bioactivity prediction.
   d. Use a multilayer perceptron regression head for MIC prediction.

```
# Pathogenic missense variants (VariPred)
# First generate wild-type/mutant embeddings, then fine-tune the MLP
# head (-t train, -b generate embeddings in batches):
python3 main.py -p <train_test_dir> -tr train -ts test_clean -t -b
# Type I anti-CRISPR: use the fixed stratified 80:20 split (random_state=41)
python3 code/training.py --model_type=CNN --feature_mode=1 \
  --esm_version=650m --run_mode=train \
  --monitor_ckpt=val_f1_score
# Peptide bioactivity (UniDL4BioPep): precompute embeddings to a unified
```

```
# CSV, MinMaxScaler-normalize, then run the notebook training cell
# (ESM_CNN, SGD with step-decay LR, ~200 epochs, early stopping,
# 10-fold cross-validation).
# MIC (Deep_AMP): precompute embeddings, then fine-tune the regressor
python generate_repre.py
# ProteinAligner embeddings
python train_regressor.py --model_architecture esm --data_type esm \
--learning_rate 1e-5 --epochs 200 # Adam, MSE, linear LR decay
```

6. Evaluate the best checkpoint on the held-out test set.
   a. Use precision, recall, and F1 score for pathogenic missense variants.
   b. Use accuracy, F1 score, and area under the ROC curve for thermostability where applicable.
   c. Use accuracy and F1 score for anti-CRISPR detection.
   d. Use the task-appropriate classification metrics for peptide bioactivity tasks.
   e. Use mean squared error for MIC prediction.

```
# Pathogenic missense variants (VariPred): predict on the held-out set
# and report precision/recall/F1 (-i input set, -o output file):
python3 main.py -p <target_dir> -i target -o VariPred_output
# Type I anti-CRISPR (AcrTransAct): evaluate the saved checkpoint;
# reports accuracy and F1 on the test split.
python3 code/training.py --model_type=CNN --feature_mode=1 \
    --esm_version=650m --run_mode=test
# Peptide bioactivity (UniDL4BioPep): the task notebook's evaluation
# cell reports ACC, BACC, Sn, Sp, MCC, and AUC on the test set.
# MIC (Deep_AMP): the training script reports test-set MSE on the
# matching regressor_test_*.csv file at the end of each run.
```

7. Repeat the run with multiple random seeds and summarize the reported metrics.
   a. Run each task with five model-training seeds while keeping the train–test membership fixed.
   b. Save the best checkpoint and test metrics from each run.
   c. Calculate the mean and standard deviation of every reported metric across the five runs.

   Note: Add a task-specific subprotocol if an application requires instructions beyond this generic workflow.

Applying ProteinAligner to a downstream task: MIC prediction (Deep_AMP)

Timing: 4–12 h depending on local device.

This worked example shows how to drop the pretrained ProteinAligner sequence encoder into the original Deep_AMP repository and obtain the final MIC-prediction results. The same pattern —

(i) clone, (ii) swap the ESM encoder weights, (iii) precompute embeddings, (iv) fit a small task head, (v) evaluate — transfers directly to the other downstream tasks.

Note: Throughout this worked example, <proteinaligner_ckpt> denotes the user's local path to the downloaded pretrained ProteinAligner checkpoint. This is the only user-specific path placeholder; all Deep_AMP split and output paths below are literal.

8. Set up the Deep_AMP repository.
   a. Clone the upstream repository.
   b. Change to the repository root.
   c. Install the repository requirements and the additional embedding dependencies.

```
git clone https://github.com/amirpandi/Deep_AMP

cd Deep_AMP

pip install -r requirements.txt

pip install fair-esm tqdm
```

Note: The repository provides regressor_train_AMP_gr_neg.csv and regressor_train_AMP_gr_pos.csv under data/train_data/AMP/. Both files contain sequence and value columns.

Note: The gr_neg file contains the Gram-negative E. coli data used in this protocol; gr_neg does not indicate a negative growth rate. The gr_pos file contains a separate Gram-positive dataset and is not used here.

9. Prepare the train/test split.
   a. Load regressor_train_AMP_gr_neg.csv.
   b. Randomly divide its 4,619 examples into an 80:20 train/test split.
      i. Set test_size to 0.2.
      ii. Set random_state to 42 and enable shuffling.
   c. Save the 3,695 training examples and 924 test examples in data/train_data/AMP/proteinaligner_split/.

Note: The Deep_AMP repository provides the Gram-negative E. coli data as one training file and does not define a test split.

⚠ CRITICAL: Keep this split fixed for every representation model and all five model-training seeds. Write the derived files to the new subdirectory; do not modify the source CSV.

```
from pathlib import Path

import pandas as pd

from sklearn.model_selection import train_test_split

source_csv = Path("data/train_data/AMP/regressor_train_AMP_gr_neg.csv")

split_dir = Path("data/train_data/AMP/proteinaligner_split")

split_dir.mkdir(parents=True, exist_ok=True)

df = pd.read_csv(source_csv)
```

```
train, test = train_test_split(
  df, test_size=0.2, random_state=42, shuffle=True
)
train_csv = split_dir / "regressor_train_AMP_gr_neg_seed42.csv"
test_csv = split_dir / "regressor_test_AMP_gr_neg_seed42.csv"
assert source_csv not in (train_csv, test_csv)
train.to_csv(train_csv, index=False)
test.to_csv(test_csv, index=False)
print(f"train={len(train)}, test={len(test)}") # train=3695, test=924
```

10. Generate ProteinAligner embeddings.
    a. Create generate_repre.py at the repository root.
    b. Load the ESM-2 650M backbone.
    c. Replace the backbone weights with the pretrained ProteinAligner weights.
    d. Mean-pool the last-layer token embeddings for each sequence.
    e. Write one embedding CSV for each split file.

```
# generate_repre.py
import torch, esm, sys, pandas as pd
from copy import deepcopy
from tqdm import tqdm
# (a) Load the ESM-2 650M backbone
model, alphabet = esm.pretrained.esm2_t33_650M_UR50D()
batch_converter = alphabet.get_batch_converter()
# (b) Overwrite with the pretrained ProteinAligner weights
ckpt = torch.load("<proteinaligner_ckpt>", map_location="cuda")
prefix = "model."
state = {k.replace(prefix, ""): deepcopy(v) for k, v in ckpt.items()
  if prefix in k and k.replace(prefix, "") in model.state_dict()}
model.load_state_dict(state, strict=False)
model = model.cuda().eval()
# (c) Per-sequence embedding: mean-pool the last-layer (33) tokens
def esm_embeddings(seq_list):
  _, _, tokens = batch_converter(seq_list)
  lens = (tokens != alphabet.padding_idx).sum(1)
  with torch.no_grad():
    out = model(tokens.cuda(), repr_layers=[33])["representations"][33]
  return pd.DataFrame([out[i, 1:lens[i]-1].mean(0).tolist() # drop BOS/EOS
    for i in range(len(lens))])
```

```
# (d) Run over a split file and save the embedding table

in_csv, out_csv = sys.argv[1], sys.argv[2]

df = pd.read_csv(in_csv)

rows = pd.DataFrame()

for seq in tqdm(df["sequence"]):

rows = pd.concat([rows, esm_embeddings([(seq, seq)])])

rows.to_csv(out_csv, index=False)
```

f. Run the script for the two derived Gram-negative E. coli seed-42 split files:

```
mkdir -p esm_representation

for split in regressor_train_AMP_gr_neg_seed42 regressor_test_AMP_gr_neg_seed42; do

python generate_repre.py data/train_data/AMP/proteinaligner_split/${split}.csv

  esm_representation/${split}.csv

done
```

Note: Each output CSV contains one row per sequence and 1,280 columns, corresponding to the ESM-2 650M embedding width.

11. Add an embedding-based regression head.
    a. Create train_regressor_esm.py at the repository root.
    b. Load the ProteinAligner embeddings and their MIC labels.
    c. Define the multilayer perceptron regression head.
    d. Train the head with mean squared error loss.
    e. Calculate MSE, R^2, and Spearman correlation on the test set and save the model state.

Note: The upstream code/train_regressor.py expects one-hot sequence input. The new entry point below consumes the ProteinAligner embeddings instead.

```
# train_regressor_esm.py

import argparse, numpy as np, pandas as pd, torch

import torch.nn as nn, torch.optim as optim

from torch.utils.data import DataLoader, TensorDataset

from sklearn.metrics import mean_squared_error, r2_score

from scipy.stats import spearmanr

class MyRegressor(nn.Module): # MLP head

def __init__(self, hidden_size=1280):

  super().__init__()

  self.fc1 = nn.Linear(hidden_size, 256)

  self.relu = nn.ReLU()
```

```
  self.fc2 = nn.Linear(256, 1)
def forward(self, x):
  return self.fc2(self.relu(self.fc1(x)))
def load_split(embed_csv, label_csv):
  X = pd.read_csv(embed_csv).to_numpy(dtype="float32")
  y = pd.read_csv(label_csv)["value"].to_numpy(dtype="float32")
  return X, y
# Code continues in the following text box.
```

```
# Continued from the preceding text box.
if __name__ == "__main__":
  p = argparse.ArgumentParser()
  p.add_argument("--train_embed", required=True)
  p.add_argument("--train_label", required=True)
  p.add_argument("--test_embed", required=True)
  p.add_argument("--test_label", required=True)
  p.add_argument("--epochs", type=int, default=200)
  p.add_argument("--lr", type=float, default=1e-5)
  p.add_argument("--batch", type=int, default=50)
  p.add_argument("--seed", type=int, default=0)
  p.add_argument("--save", default="regressor.pt")
  args = p.parse_args()
  torch.manual_seed(args.seed); np.random.seed(args.seed)
  Xtr, ytr = load_split(args.train_embed, args.train_label)
  Xte, yte = load_split(args.test_embed, args.test_label)
  loader = DataLoader(TensorDataset(torch.tensor(Xtr), torch.tensor(ytr)),
    batch_size=args.batch, shuffle=True)
  model = MyRegressor(hidden_size=Xtr.shape[1]).cuda()
  opt = optim.Adam(model.parameters(), lr=args.lr)
  sched = optim.lr_scheduler.LambdaLR(opt, lambda e: 1 - e / args.epochs)
  loss_fn = nn.MSELoss()
  for epoch in range(args.epochs): # training loop
    model.train()
    for xb, yb in loader:
      xb, yb = xb.cuda(), yb.cuda().unsqueeze(1)
      opt.zero_grad()
      loss = loss_fn(model(xb), yb)
      loss.backward(); opt.step()
```

```
    sched.step()
  model.eval() # evaluation
  with torch.no_grad():
    pred = model(torch.tensor(Xte).cuda()).cpu().numpy().squeeze()
  print(f"MSE = {mean_squared_error(yte, pred):.4f}")
  print(f"R^2 = {r2_score(yte, pred):.4f}")
  print(f"Spearman = {spearmanr(yte, pred).correlation:.4f}")
torch.save(model.state_dict(), args.save)
```

12. Fine-tune the regression head.
    a. Use the two embedding CSVs generated in step 10 as the training and test features.
    b. Use the corresponding fixed-split CSVs generated in step 9 as the label files.
    c. Run the regression script for 200 epochs with a learning rate of 1×10^{-5}.
    d. Save the fitted weights as regressor_neg.pt.

    Note: The fixed-split files provide the value column, and the embedding files provide their corresponding ProteinAligner features.

    ⚠ CRITICAL: Use only the Gram-negative E. coli seed-42 files. Do not replace or overwrite the repository's original training CSV.

```
python train_regressor_esm.py \
  --train_embed esm_representation/regressor_train_AMP_gr_neg_seed42.csv \
  --train_label data/train_data/AMP/proteinaligner_split/
  regressor_train_AMP_gr_neg_seed42.csv \
  --test_embed esm_representation/regressor_test_AMP_gr_neg_seed42.csv \
  --test_label data/train_data/AMP/proteinaligner_split/
  regressor_test_AMP_gr_neg_seed42.csv \
--epochs 200 --lr 1e-5 --save regressor_neg.pt
```

13. Evaluate the saved checkpoints.
    a. Read the test-set MSE, R^2, and Spearman correlation printed at the end of each run.
    b. Report these values as the MIC-prediction metrics.

    Note: Calculate the MSE for each run on the fixed seed-42 held-out Gram-negative E. coli test set created in step 9. Use the same 924 examples for every representation model and model-training seed; do not combine Gram-positive and Gram-negative results.

14. Repeat the training across seeds and aggregate the results.
    a. Run steps 12–13 with –seed 0,1,2,3,4.
    b. Calculate the mean and standard deviation of each metric across the five runs.

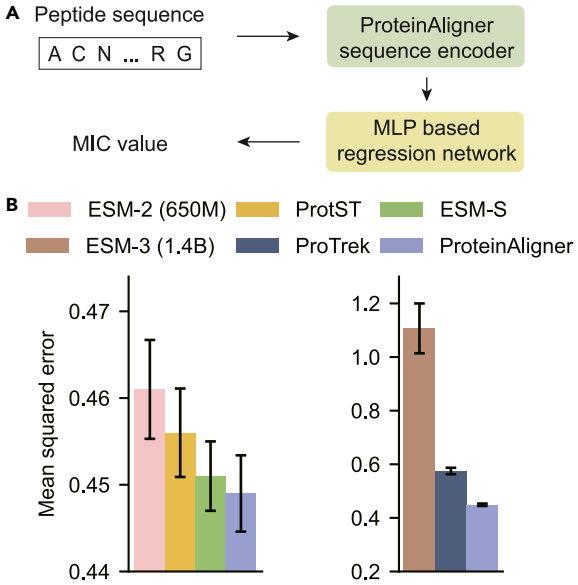


Figure 1. MIC-prediction results on the Deep_AMP benchmark

(A) Inference pipeline. A peptide amino-acid sequence is encoded by the pretrained ProteinAligner sequence encoder (with ESM-2 650M backbone weights overwritten by the ProteinAligner checkpoint), and the resulting mean-pooled embedding is passed to an MLP-based regression head that predicts the minimum inhibitory concentration (MIC) value.

(B) Test-set mean squared error of ProteinAligner versus five protein-language-model baselines (ESM-2 650M, ProtST, ESM-S, ESM-3 1.4B, and ProTrek) on the fixed 80:20 held-out E. coli (Gram-negative) Deep_AMP split created with random_state=42 (3,695 training examples and 924 test examples). This same split was used for all methods and all five model-training seeds. Results are reported as mean ± standard deviation over five independent runs: ESM-2 650M, 0.461 ± 0.006; ProtST, 0.456 ± 0.005; ESM-S, 0.451 ± 0.004; ESM-3 1.4B, 1.107 ± 0.093; ProTrek, 0.575 ± 0.012; and ProteinAligner, 0.449 ± 0.004. Error bars denote one standard deviation. The y-axis is split into two ranges: the left sub-panel zooms in on the three lower-error baselines together with ProteinAligner (range 0.44–0.47), while the right sub-panel preserves the absolute scale of the two higher-error baselines together with ProteinAligner (range 0.2–1.2). ProteinAligner attains the lowest mean MSE among the evaluated models.

Note: These seeds change only model initialization and training stochasticity. The seed-42 train/test membership remains fixed.

```
for s in 0 1 2 3 4; do
  python train_regressor_esm.py \
    --train_embed esm_representation/regressor_train_AMP_gr_neg_seed42.csv \
    --train_label data/train_data/AMP/proteinaligner_split/
    regressor_train_AMP_gr_neg_seed42.csv \
    --test_embed esm_representation/regressor_test_AMP_gr_neg_seed42.csv \
    --test_label data/train_data/AMP/proteinaligner_split/
    regressor_test_AMP_gr_neg_seed42.csv \
    --seed $s --save regressor_neg_seed${s}.pt \
|  tee log_neg_seed${s}.txt
done
```

Note: Step 9 writes the derived split under data/train_data/AMP/proteinaligner_split/ without overwriting the source CSV. It contains the seed-42 training and test CSVs specified in step 9.

Note: The five runs save regressor_neg_seedN.pt and log_neg_seedN.txt, where N is 0, 1, 2, 3, or 4. Each log reports test-set MSE, R2, and Spearman correlation. Together, these files constitute the final MIC results for ProteinAligner on Deep_AMP.

Note: Figure 1 summarizes the MIC-prediction outcome on the Deep_AMP benchmark. Panel (a) shows the inference pipeline assembled in steps 8–14: a peptide sequence is fed through the pretrained ProteinAligner sequence encoder to obtain a mean-pooled embedding, which the MLP-based regression head then maps to a predicted MIC value.

EXPECTED OUTCOMES

Successful execution of this protocol should produce (1) processed manifests for multimodal pretraining, (2) a pretrained ProteinAligner checkpoint with stable contrastive-loss curves, and (3) downstream-task checkpoints and metric summaries for the selected application. Expected outputs include reusable sequence or structure embeddings, task-specific prediction scores, and per-seed evaluation tables.

Note: The reference pretraining trajectory used two corpora: 133,726 sequence–structure pairs and 290,480 sequence–text pairs. It used random seed 43, a maximum of 64 epochs, perprocess batch size 2, one gradient-accumulation step, learning rate $5 \times 10{-}6$, weight decay 0.05, AdamW betas (0.9, 0.95), gradient clipping at 1.0, InfoNCE temperature 0.1, cosine learning-rate decay, mixed-precision training, and self-contrast disabled.

Note: The selected checkpoint was saved after logged epoch 22 (zero-indexed; 23 completed epochs). The two-GPU command above is a resource-scaled example. To reproduce the reference trajectory, set –epochs 64 and retain the checkpoint saved after logged epoch 22.

Note: The representative pretraining-log excerpt below uses dual_nll for the sequence– structure loss and dual_nll_2 for the sequence–text loss. The first total-loss value is the median over the logger's 20-iteration smoothing window, and the parenthesized value is the cumulative average for the epoch.

```
Epoch: [0] [0/3595] lr: 0.000005
dual_nll: 6.393367 dual_nll_2: 6.284734 loss: 12.6781 (12.6781)
Epoch: [0] [3594/3595] lr: 0.000005
dual_nll: 0.126235 dual_nll_2: 0.323230 loss: 0.3325 (0.7444)
Epoch: [11] [0/3595] lr: 0.000005
dual_nll: 0.009822 dual_nll_2: 0.127505 loss: 0.1373 (0.1373)
Epoch: [11] [3594/3595] lr: 0.000005
dual_nll: 0.006079 dual_nll_2: 0.106224 loss: 0.1193 (0.1464)
Epoch: [22] [0/5991] lr: 0.000005
dual_nll: 0.104863 dual_nll_2: 0.156478 loss: 0.2613 (0.2613)
Epoch: [22] [5990/5991] lr: 0.000005
dual_nll: 0.008065 dual_nll_2: 0.093977 loss: 0.1166 (0.1361)
```

Note: At the values printed every 20 iterations (plus the final iteration), the smoothed total loss ranged from 0.2980 to 12.6781 in epoch 0, from 0.0939 to 0.1917 in epoch 11, and from 0.0541 to 0.2613 in epoch 22. The corresponding end-of-epoch global-average losses were 0.7444, 0.1464, and 0.1361, respectively. These values are diagnostic reference points: small numerical differences are expected across hardware and software environments, whereas failure to show the large initial decrease or subsequent stabilization may indicate an abnormal run.

QUANTIFICATION AND STATISTICAL ANALYSIS

All statistical analyses in the source manuscript were performed in Python using scipy.stats. Unless otherwise noted, each downstream experiment was repeated for five independent runs with different random seeds and no cross-validation. The reported summary statistic was the mean across runs, and variability was reported as standard deviation. Statistical significance was assessed with two-sided paired t-tests on per-seed task metrics, with a significance threshold of $p < 0.05$ and no multiple-comparison correction. Each downstream benchmark was analyzed as a separate task-specific comparison rather than as part of a prespecified confirmatory family of hypotheses. Nevertheless, because no false-discovery-rate correction was applied across the collection of downstream comparisons, the reported p-values represent nominal, unadjusted evidence. They should not be interpreted as demonstrating significance controlled across all benchmarks, and nominally significant comparisons, particularly those with p-values close to 0.05, may not remain significant after false-discovery-rate correction. Official train–test splits were used when available; otherwise, a fixed-seed random split was applied. For Deep_AMP, every method and all five model-training seeds used the same 80:20 Gram-negative E. coli split created with random_state=42 (3,695 training examples and 924 test examples). For type I anti- CRISPR activity detection, all methods and model-training seeds likewise used the same stratified 80:20 split created with random_state=41. Repeating model training across random seeds measures variability due to initialization and stochastic optimization while holding the train–test partition fixed; it does not quantify sensitivity to alternative data partitions. Neither independent external validation nor nested cross-validation was performed for these two tasks.

LIMITATIONS

ProteinAligner depends on the availability and quality of paired protein sequences, structures, and text annotations. Even though the framework can be trained with incomplete modality coverage, missing structural or textual information may limit representation quality. The protocol is also computationally expensive, especially during distributed pretraining on large public corpora. Finally, task-specific source repositories and local filesystem layouts may evolve, so users should record repository versions and resolve the explicitly defined user-supplied path tokens for their environment.

TROUBLESHOOTING

Problem 1

The pretraining manifests contain duplicate identifiers, missing mappings, or inaccessible structure files (related to step 7 in "before you begin").

Potential solution

Validate the sequence, structure, and text identifiers before training. Check the sequence–text and sequence–structure mappings separately. Generate an audit table containing the sequence identifier, structure identifier, text identifier, and source path for every row. Resolve or exclude every duplicate, missing mapping, and inaccessible file before launching pretraining.

Problem 2

Distributed pretraining is unstable, exceeds GPU memory, or produces inconsistent worker behavior (related to step 2 in "multimodal pretraining of ProteinAligner").

Potential solution

Reduce the per-device batch size. Increase gradient accumulation if needed to preserve the effective global batch size. Inspect the mixed-precision and gradient-accumulation settings, and confirm that every worker can read the same processed manifests. Record every deviation from the published 40-A100 configuration.

Problem 3

Fine-tuning metrics vary strongly across model-training seeds (related to step 7 in "generic fine-tuning workflow for downstream prediction" and step 14 in "applying ProteinAligner to a downstream task: MIC prediction (Deep_AMP)").

Potential solution

Keep the train–test split fixed across runs. Repeat training with five model-training seeds, compare the validation trajectories, and retain the best checkpoint according to the predefined stopping rule. Report the mean and standard deviation of each test metric across the five runs.

RESOURCE AVAILABILITY

Lead contact

Requests for further information and resources should be directed to and will be fulfilled by the lead contact, Pengtao Xie (p1xie@ucsd.edu).

Technical contact

Technical questions about implementation and reproducibility should be directed to Pengtao Xie (p1xie@ucsd.edu).

Materials availability

This study did not generate new physical materials.

Data and code availability

The FASTA and PDB source data are publicly available through UniProtKB/Swiss-Prot and RCSB PDB. ProteinAligner v1.1, containing the source code, pretraining mapping manifests, pretrained checkpoint, and SHA-256 checksum, is archived at https://doi.org/10.5281/zenodo.21891297 and corresponds to Git commit 3bb238f8; the full commit is specified in the setup command above. The concept DOI https://doi.org/10.5281/zenodo.18806818 resolves to the latest archived version. The downstream repositories and exact commits are reported in the key resources table. Any additional information required to reanalyze the data reported in the original study is available from the lead contact upon request.

ACKNOWLEDGMENTS

This work was funded by P.X. via grant NSF IIS2405974, NSF IIS2339216, and NIH R35GM157217. The authors thank all members of the lab for their support.

AUTHOR CONTRIBUTIONS

Conceptualization, L.Z., H.G., and P.X.; methodology, L.Z., H.G., and P.X.; investigation, L.Z., H.G., E.X., and P.X.; writing – original draft, L.Z., H.G., and P.X.; writing – review and editing, L.Z., H.G., L.S., Y.S.K., D.S., H.R., D.G., E.V., M.G., W.W., T.I., E.X., and P.X.; funding acquisition, P.X.; resources, E.X. and P.X.; supervision, E.X. and P.X.

DECLARATION OF INTERESTS

T.I. is a cofounder, a member of the advisory board, and has an equity interest in Data4Cure and Serinus Biosciences.

DECLARATION OF GENERATIVE AI AND AI-ASSISTED TECHNOLOGIES IN THE WRITING PROCESS

During the preparation of this work, the authors used ChatGPT (OpenAI) to polish the writing and correct grammatical errors. After using this tool, the authors reviewed and edited the content as needed and take full responsibility for the content of the publication.

REFERENCES

1. Zhang, L., Guo, H., Schaffer, L., Ko, Y.S., Singh, D., Rahmani, H., Grotjahn, D., Villa, E., Gilson, M., Wang, W., et al. (2026). A tri-modal contrastive learning framework for protein representation learning. Cell Rep. Methods *6*, 101407. https://doi.org/10.1016/j.crmeth.2026.101407.

2. Lin, Z., Akin, H., Rao, R., Hie, B., Zhu, Z., Lu, W., Smetanin, N., Verkuil, R., Kabeli, O., Shmueli, Y., et al. (2023). Evolutionary-scale prediction of atomic-level protein structure with a language model. Science *379*, 1123–1130. https://doi.org/10.1126/science.ade2574.

3. Rives, A., Meier, J., Sercu, T., Goyal, S., Lin, Z., Liu, J., Guo, D., Ott, M., Zitnick, C.L., Ma, J., and Fergus, R. (2021). Biological structure and function emerge from scaling unsupervised learning to 250 million protein sequences. Proc. Natl. Acad. Sci. USA *118*, e2016239118. https://doi.org/10.1073/pnas.2016239118.

4. Xie, D., Yu, X., Zeng, Y., and Dai, Q. (2026). MFDR-DTI: multisource feature decoupling and dual-level reorganization fusion for accurate drug–target interaction prediction. ACS Omega *11*, 19641–19657. https://doi.org/10.1021/acsomega.5c13460.

5. Xiong, P., Chen, H., Zhou, J., Zeng, Y., and Dai, Q. (2025). MGRL-DDI: multiview graph representation learning for accurate drug–drug interaction prediction. J. Chem. Inf. Model. *65*, 9800–9814. https://doi.org/10.1021/acs.jcim.5c01773.

6. Chen, H., Qian, Y., Dai, Q., Zheng, Y.-F., Zheng, Z.-C., and Shi, Z.-X. (2026). Analyzing cell-type-specific isoform expression using IsoDiffR and long-read single-cell RNA sequencing. Bioinformatics *42*, btaf664. https://doi.org/10.1093/bioinformatics/btaf664.

7. Li, J., Zhang, W., and Dai, Q. (2025). scDGG: dynamic gene graphs for enhancing clustering analysis of single-cell RNA sequencing data via spatiotemporal representations. Comput. Biol. Chem. *118*, 108483. https://doi.org/10.1016/j.compbiolchem.2025.108483.

8. Dai, Q., Chen, H., Yi, W.-J., Zhao, J.-N., Zhang, W., He, P.-A., Liu, X.-Q., Zheng, Y.-F., and Shi, Z.-X. (2024). Precision DNA methylation typing via hierarchical clustering of nanopore current signals and attention-based neural network. Brief. Bioinform. *25*, bbae596. https://doi.org/10.1093/bib/bbae596.

9. Guo, X., Guo, Y., Chen, H., Liu, X., He, P., Li, W., Zhang, M.Q., and Dai, Q. (2023). Systematic comparison of genome information processing and boundary recognition tools used for genomic island detection. Comput. Biol. Med. *166*, 107550. https://doi.org/10.1016/j.compbiomed.2023.107550.

10. Yang, Z., Yi, W., Tao, J., Liu, X., Zhang, M.Q., Chen, G., and Dai, Q. (2022). HPVMD-C: a disease-based mutation database of human papillomavirus in China. Database *2022*, baac018. https://doi.org/10.1093/database/baac018.

11. Lee, J., Yoon, W., Kim, S., Kim, D., Kim, S., So, C.H., and Kang, J. (2020). BioBERT: a pre-trained biomedical language representation model for biomedical text mining. Bioinformatics *36*, 1234–1240. https://doi.org/10.1093/bioinformatics/btz682.

12. Petsko, G.A., and Ringe, D. (2004). Protein Structure and Function (New Science Press).

13. Xu, M., Yuan, X., Miret, S., and Tang, J. (2023). ProtST: multi-modality learning of protein sequences and biomedical texts. In International Conference on Machine Learning (PMLR), pp. 38749–38767.

14. UniProt Consortium (2019). UniProt: a worldwide hub of protein knowledge. Nucleic Acids Res. *47*, D506–D515. https://doi.org/10.1093/nar/gky1049.

15. Burley, S.K., Bhikadiya, C., Bi, C., Bittrich, S., Chao, H., Chen, L., Craig, P.A., Crichlow, G.V., Dalenberg, K., Duarte, J.M., et al. (2023). RCSB Protein Data Bank (RCSB.org): delivery of experimentally-determined PDB structures alongside one million computed structure models of proteins from artificial intelligence/machine learning. Nucleic Acids Res. *51*, D488–D508. https://doi.org/10.1093/nar/gkac1077.

16. Lin, W., Wells, J., Wang, Z., Orengo, C., and Martin, A.C.R. (2024). Enhancing missense variant pathogenicity prediction with protein language models using VariPred. Sci. Rep. *14*, 8136. https://doi.org/10.1038/s41598-024-51489-7.

17. Chen, T., Gong, C., Diaz, D.J., Chen, X., Wells, J.T., Liu, Q., Wang, Z., Ellington, A., Dimakis, A., and Klivans, A. (2023). HotProtein: a novel framework for protein thermostability prediction and editing. In International Conference on Learning Representations (OpenReview.net).

18. Hasani, M., Trost, C.N., Timmerman, N., and Jin, L. (2023). AcrTransAct: pretrained protein transformer models for the detection of type I anti-CRISPR activities. In Proceedings of the 14th ACM Conference on Bioinformatics, Computational Biology, and Health Informatics (ACM), p. 6.

19. Du, Z., Ding, X., Xu, Y., and Li, Y. (2023). UniDL4BioPep: a universal deep learning architecture for binary classification in peptide bioactivity. Brief. Bioinform. *24*, bbad135. https://doi.org/10.1093/bib/bbad135.

20. Pandi, A., Adam, D., Zare, A., Trinh, V.T., Schaefer, S.L., Burt, M., Klabunde, B., Bobkova, E., Kushwaha, M., Foroughijabbari, Y., et al. (2023). Cell-free biosynthesis combined with deep learning accelerates de novo development of antimicrobial peptides. Nat. Commun. *14*, 7197. https://doi.org/10.1038/s41467-023-42434-9.

21. Hsu, C., Verkuil, R., Liu, J., Lin, Z., Hie, B., Sercu, T., Lerer, A., and Rives, A. (2022). Learning inverse folding from millions of predicted structures. In International Conference on Machine Learning (PMLR), pp. 8946–8970.