EECS 182 Deep Neural Networks
Fall 2025 Anant Sahai and Gireeja Ranade

Homework 7

This homework is due on Mar 30, at 10:59PM.

1. Implementing RNNs (Coding)

This problem involves filling out this notebook.

(a) **Implement Section 1A in the notebook**, which constructs a vanilla RNN layer. This layer implements the function

$$h_t = \sigma(W^h h_{t-1} + W^x x_t + b)$$

where W^h, W^x, and b are learned parameter matrices, x is the input sequence, and σ is a nonlinearity such as tanh. The RNN layer "unrolls" across a sequence, passing a hidden state between timesteps and returning an array of hidden states at all timesteps.

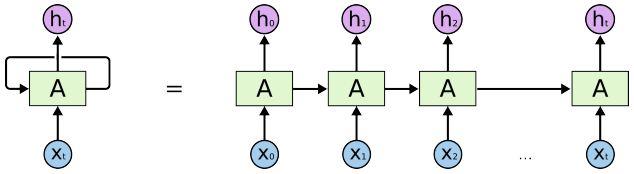


Figure 1: Source: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

Copy the outputs of the *"Test Cases"* code cell and paste it into your submission of the written assignment.

Solution: See the solution notebook. Each max error should be less than 1e-4.

(b) **Implement Section 1.B of the notebook**, in which you'll use this RNN layer in a regression model by adding a final linear layer on top of the RNN outputs.

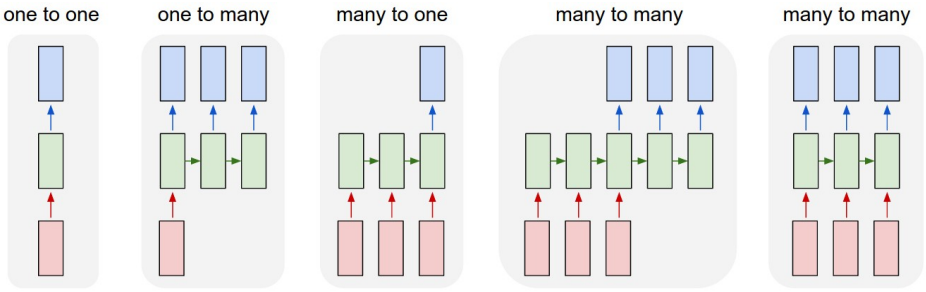
$$\hat{y}_t = W^f h_t + b^f$$

We'll compute one prediction for each timestep.

Copy the outputs of the *"Tests"* code cell and paste it into your submission of the written assignment.

Solution: See the solution notebook. Each max error should be less than 1e-4.

(c) RNNs can be used for many kinds of prediction problems, as shown below. In this notebook we will look at many-to-one prediction and aligned many-to-many prediction.



We will use a simple averaging task. The input X consists of a sequence of numbers, and the label y is a running average of all numbers seen so far.

We will consider two tasks with this dataset:

- Task 1: predict the running average at all timesteps
- Task 2: predict the average at the last timestep only

Implement Section 1.C in the notebook, in which you'll look at the synthetic dataset shown and implement a loss function for the two problem variants.

Copy the outputs of the *"Tests"* code cell and paste it into your submission of the written assignment.

Solution: See the solution notebook. Each max error should be less than `1e-4`.

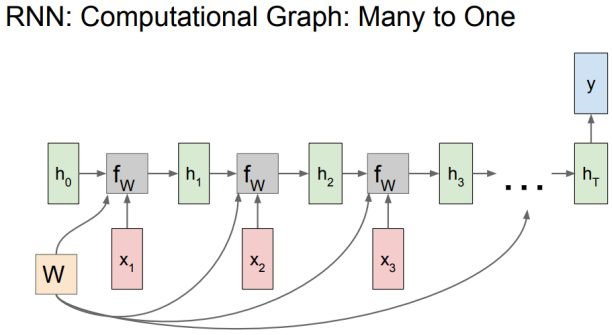


Figure 2: Image source: https://calvinfeng.gitbook.io/machine-learning-notebook/supervised-learning/recurrent-neural-network/recurrent_neural_networks

(d) Consider an RNN which outputs a single prediction at timestep T. As shown in Figure 2, each weight matrix W influences the loss by multiple paths. As a result, the gradient is also summed over multiple paths:

$$\frac{\partial \mathcal{L}}{\partial W} = \frac{\partial \mathcal{L}}{\partial h_T}\frac{\partial h_T}{\partial W} + \frac{\partial \mathcal{L}}{\partial h_{T-1}}\frac{\partial h_{T-1}}{\partial W} + \ldots + \frac{\partial \mathcal{L}}{\partial h_1}\frac{\partial h_1}{\partial W} \tag{1}$$

When you backpropagate a loss through many timesteps, the later terms in this sum often end up with either very small or very large magnitude - called vanishing or exploding gradients respectively. Either problem can make learning with long sequences difficult.

Implement Notebook Section 1.D, which plots the magnitude at each timestep of $\frac{\partial \mathcal{L}}{\partial h_t}$. Play around with this visualization tool and try to generate exploding and vanishing gradients.

Include a screenshot of your visualization in the written assignment submission.

Solution: See the solution notebook.

(e) **If the network has no nonlinearities, under what conditions would you expect the exploding or vanishing gradients with for long sequences? Why?** (Hint: it might be helpful to write out the formula for $\frac{\partial \mathcal{L}}{\partial h_t}$ and analyze how this changes with different t). **Do you see this pattern empirically using the visualization tool in Section 1.D in the notebook** with last_step_only=True?

Solution: If we use the MSE loss on a single example (x, y), the gradient $\frac{\partial \mathcal{L}}{\partial h_t} = 2(\hat{y}-y)W^f(W^h)^{T-t}$. (To clarify, the exponents f and h are matrix indicators, but $T-t$ is an exponent.) If the magnitude of the largest eigenvalue of W^h is much greater than 1, the gradient will explode, and if it's much less than 1, the gradient will start to vanish. Gradients are only stable when the largest eigenvalue magnitude is close to 1.

We see the expected pattern empirically.

(f) Compare the magnitude of hidden states and gradients when using ReLU and tanh nonlinearities in Section 1.D in the notebook. **Which activation results in more vanishing and exploding gradients? Why?** (This does not have to be a rigorous mathematical explanation.)

Solution: Hidden states: tanh restricts hidden state values to (-1, 1), so hidden state magnitudes remain small. With ReLU, hidden state values can easily explode.

Gradients: When tanh inputs are large, gradients are close to zero. This results in fewer exploding gradients but more vanishing gradients. Exploding gradients are still possible, however, when the largest eigenvalue of W_h has magnitude > 1, but the hidden states remain close to zero. ReLU activations, in contrast, result in frequent exploding hidden state sizes and gradients, similar to in the no-activation RNN.

(g) **What happens if you set last_target_only = False in Section 1.D in the notebook? Explain why this change affects vanishing gradients. Does it help the network's ability to learn dependencies across long sequences?** (The explanation can be intuitive, not mathematically rigorous.)

Solution: For every timestep k, the model's prediction produces loss $\mathcal{L}_k$. The gradient $\partial L_k/\partial h_k$ is high-magnitude, resulting in high-magnitude logged gradients in the visualization tool, but for any timestep $t << k$, $\partial L_k/\partial h_t$ will still be small. This means the network will still struggle to pass gradients over long sequences, making it hard to learn long-range dependencies.

(h) (Optional) We can create multi-layer recurrent networks by stacking layers as shown in Figure 3. The hidden state outputs from one layer become the inputs to the layer above.

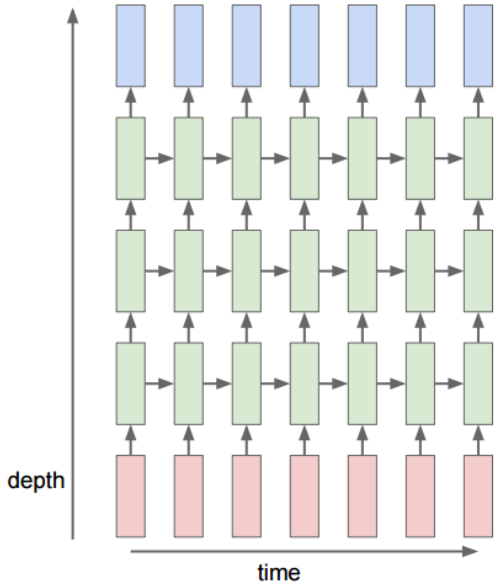


Figure 3: Image source: https://calvinfeng.gitbook.io/machine-learning-notebook/supervised-learning/recurrent-neural-network/recurrent_neural_networks

Implement notebook Section 1.K and run the last cell to train your network. You should be able to reach training loss < 0.001 for the 2-layer networks, and <.01 for the 1-layer networks.

2. RNNs for Last Name Classification (Coding)

Please follow the instructions in this notebook. You will train a neural network to predict the probable language of origin for a given last name / family name in Latin alphabets.

(a) Although the neural network you have trained is intended to predict the language of origin for a given last name, it could potentially be misused. **In what ways do you think this could be problematic in real-world applications**?

Solution: The model's predictions could be used to make assumptions about a person's nationality or ethnicity, which could lead to discrimination or bias. It's important to note that the model should only be used to make predictions about the language of origin for a given name and not to make assumptions about a person's identity. The model could also be used to infer personal information about individuals, such as their cultural background, which could be considered an invasion of privacy.

3. Auto-encoder : Learning without Labels

So far, with supervised learning algorithms we have used labelled datasets $\mathcal{D} = \{\mathbf{X}, \mathbf{y}\}$ to learn a mapping f_θ from input x to labels y. In this problem, we now consider algorithms for *unsupervised learning*, where we are given only inputs x, but no labels y. At a high-level, unsupervised learning algorithms leverage some structure in the dataset to define proxy tasks, and learn *representations* that are useful for solving downstream tasks.

Autoencoders present a family of such algorithms, where we consider the problem of learning a function $f_\theta : \mathbb{R}^m \to \mathbb{R}^k$ from input x to a *intermediate representation* z of x (usually $k \ll m$). For autoencoders, we use reconstructing the original signal as a proxy task, i.e. representations are more likely to be useful for downstream tasks if they are low-dimensional but encode sufficient information to approximately reconstruct the input. Broadly, autoencoder architectures have two modules:

- An encoder $f_\theta : \mathbb{R}^m \to \mathbb{R}^k$ that maps input x to a representation z.
- A decoder $g_\phi : \mathbb{R}^k \to \mathbb{R}^m$ that maps representation z to a reconstruction $\hat{x}$ of x.

In such architectures, the parameters (θ, ϕ) are learnable, and trained with the learning objective of minimizing the reconstruction error $\ell(\hat{x}_i, x_i) = \|x - \hat{x}\|_2^2$ using gradient descent.

$$\theta_{\text{enc}}, \phi_{\text{dec}} = \underset{\Theta,\Phi}{\text{argmin}} \frac{1}{N} \sum_{i=1}^{N} \ell(\hat{x}_i, x_i)$$
$$\hat{x} = g_\phi(f_\theta(x))$$

Note that above optimization problem does not require labels $\mathbf{y}$. In practice however, we would want to use the *pretrained* models to solve the *downstream* task at hand, e.g. classifying MNIST digits. *Linear Probing* is one such approach, where we fix the encoder weights, and learn a simple linear classifier over the features $z = \text{encoder}(x)$.

(a) Designing AutoEncoders (Coding)

Please follow the instructions in this notebook. You will train autoencoders, denoising autoencoders, and masked autoencoders on a synthetic dataset and the MNIST dataset. Once you finished with the notebook,

- Answer the following questions in your submission of the written assignment:

(i) **Show your visualization** of the vanilla autoencoder with different latent representation sizes.
Solution:

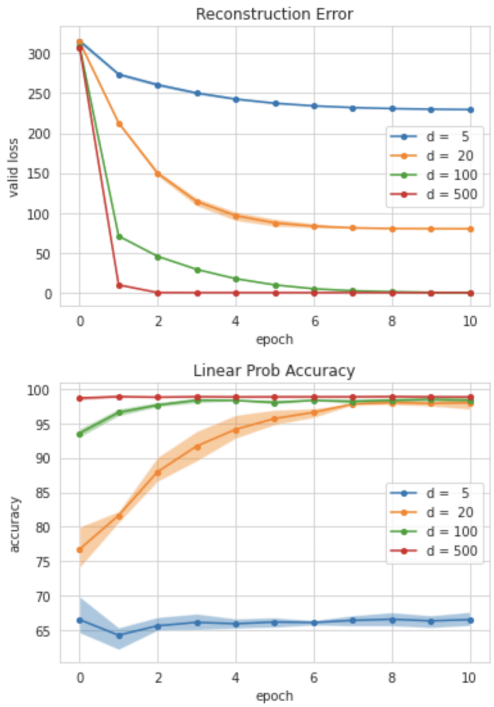


Figure 4: Visualization of the vanilla autoencoder with different latent representation sizes.

See Figure 4. Please refer to the solution notebook for the codes.

(ii) Based on your previous visualizations, answer this question: **How does changing the latent representation size of the autoencoder affect the model's performance in terms of reconstruction accuracy and linear probe accuracy? Why?**

Solution: Based on the given synthetic dataset, each data point has 100 dimensions, with 20 high-variance dimensions affecting the class label. The following observations can be made from the visualizations in Figure 4.

Firstly, the reconstruction error of the autoencoder decreases as the size of the latent representation increases. However, this reduction becomes marginal when the size of the latent representation exceeds the dimension of the data (100).

Secondly, the linear probe accuracy increases as the size of the latent representation increases. When the size of the latent representation exceeds the dimension of the data (100), the linear probe accuracy approaches $\sim$100% even without training the autoencoder. However, when the size of the latent representation equals the number of interpretive dimensions (20), training the autoencoder becomes essential for achieving a high linear probe accuracy. The linear probe accuracy finally converges to $\geq$95% with training. On the other hand, if the size of the latent representation is as small as 5, it fails to capture all useful information in the input data, leading to significantly lower linear probe accuracy.

(b) PCA & AutoEncoders

In the case where the encoder f_θ, g_ϕ are linear functions, the model is termed as a *linear autoencoder*. In particular, assume that we have data $x_i \in \mathbb{R}^m$ and consider two weight matrices: an encoder $W_1 \in \mathbb{R}^{k\times m}$ and decoder $W_2 \in \mathbb{R}^{m\times k}$ (with $k < m$). Then, a linear autoencoder learns a low-dimensional embedding of the data $\mathbf{X} \in \mathbb{R}^{m\times n}$ (which we assume is zero-centered without loss of generality) by minimizing the objective,

$$\mathcal{L}(W_1, W_2; \mathbf{X}) = ||\mathbf{X} - W_2 W_1 \mathbf{X}||_F^2 \tag{2}$$

We will assume $\sigma_1^2 > \cdots > \sigma_k^2 > 0$ are the k largest eigenvalues of $\mathbf{X}\mathbf{X}^\top$. The assumption that the $\sigma_1, \ldots, \sigma_k$ are positive and distinct ensures identifiability of the principal components, and is common in this setting. Therefore, the top-k eigenvalues of $\mathbf{X}$ are $S = \mathrm{diag}(\sigma_1, \ldots, \sigma_k)$, with corresponding eigenvectors are the columns of $\mathbf{U}_k \in \mathbb{R}^{m\times k}$. A well-established result from [1] shows that principal components are the unique optimal solution to linear autoencoders (up to sign changes to the projection directions). In this subpart, we take some steps towards proving this result.

(i) **Write out the first order optimality conditions that the minima of Eq. 2 would satisfy.**

Solution: We can compute the first order conditions for W_1 and W_2, respectively.

Important equations:

$$\|A\|_F^2 = tr(AA^T), \quad tr(ABC) = tr(BCA) = tr(CAB), \quad tr(A) = tr(A^T)$$

$$\nabla_A tr(AB) = B^T, \quad \nabla_B tr(AB) = A^T$$

Also, taking the matrix derivative of trace follows the product rule $(uv)' = u'v + uv'$. So for example (assuming all the matrices are of the right shape):

$$\nabla_A tr(ABBA) = \nabla_A tr(ABBC)|_{C=A} + \nabla_A tr(CBBA)|_{C=A} \quad \text{product rule of derivatives}$$

[1] Baldi, Pierre, and Kurt Hornik. "Neural networks and principal component analysis: Learning from examples without local minima" (1989)

$$= \nabla_A tr(ABBC)|_{C=A} + \nabla_A tr(ACBB)|_{C=A} \quad \text{cyclic rule}$$
$$= (BBA)^T + (ABB)^T$$

Now just do the calculation.

$$\begin{aligned}\nabla_{W_2} L &= \nabla_{W_2} tr(XX^T + W_2W_1XX^TW_1^TW_2^T - 2W_2W_1XX^T) \\ &= \cancel{\nabla_{W_2} tr(XX^T)} + \nabla_{W_2} tr(W_2W_1XX^TW_1^TW_2^T) - \nabla_{W_2} 2tr(W_2W_1XX^T) \quad \text{linearity} \\ &= 2(W_1XX^TW_1^TW_2^T)^T - 2(W_1XX^T)^T \\ &= 2(W_2W_1 - I)XX^TW_1^T\end{aligned}$$

Similarly calculate the other one:

$$\nabla_{W_1} L = 2W_2^T(W_2W_1 - I)XX^T$$

(ii) **Show that the principal components $\mathbf{U}_k$ satisfy the optimality conditions outlined in (i).**
Solution: Do the SVD decomposition: $X = U\Sigma V^T$, and $D = \Sigma\Sigma^T$. Note that D is a diagonal matrix.
Then the first order condition states that

$$W_2^T(W_2W_1 - I)UDU^T = 0; \quad (W_2W_1 - I)UDU^TW_1^T = 0$$

Let $V_1 := W_1U, V_2 := U^TW_2$, then it further simplifies to

$$(V_2V_1 - I)DV_1^T = 0; \quad V_2^T(V_2V_1 - I)D = 0$$

If $W_1^T = W_2$ are the first k columns of U, then

$$V_1 = \left[\begin{array}{ccc|c} 1 & & & \\ & \ddots & & 0 \\ & & 1 & \end{array}\right] \tag{3}$$

and $V_2 = V_1^T$. Now direct computation finishes the proof.

4. Read a Blog Post: How to train your Resnet

In previous homeworks, we saw how memory and compute constraints on GPUs put limits on the architecture and the hyperparameters (e.g., batch size) we can use to train our models. To train better models, we could scale up by using multiple GPUs, but most distributed training techniques scale sub-linearly and often we simply don't have as many GPU resources at our disposal. This raises a natural question - how can we make model training more efficient on a single GPU?

The blog series How to train your Resnet explores how to train ResNet models efficiently on a single GPU. It covers a range of topics, including architecture, weight decay, batch normalization, and hyperparameter tuning. In doing so, it provides valuable insights into the training dynamics of neural networks and offers lessons that can be applied in other settings.

Read the blog series and answer the questions below.

(a) **What is the baseline training time and accuracy the authors started with? What was the final training time and accuracy achieved by the authors?**

Solution: The baseline was Ben Johnson's run, which reached 94% accuracy in 341s. With the optimizations made by the authors, they achieved 94.1% in 26s.

(b) **Comment on what you have learnt.** (≈ 100 words)

Solution: There is no "right" answer. This is an open ended question, and each student may have a different answer. Key takeaways can include (but are not limited to) tradeoffs between small batch vs large batch training, computing batch norms can be slow if not using optimized code, the importance of resource-constraining ML benchmarks, weight-decay dynamics, the importance of batch norm etc.

(c) **Which approach taken by the authors interested you the most? Why?** (≈ 100 words)

Solution: Open ended question, there's no "right" answer.

5. The power of the graph perspective in clustering (Coding)

Implement all the TODOs in the notebook. **Answer the written questions below.**

(a) We used the KMeans algorithm implementation of sklearn, and showed our attempt to cluster this dataset into 3 classes. **Comment on the output the KMeans algorithm. Did it work? If so, explain why, if not, explain why not.**

Solution: As could be seen in the plots, it did not work. Note that the provided dataset is not linearly separable and the KMeans clustering algorithm minimizes within-cluster variances (squared Euclidean distances), but not regular Euclidean distances.

This algorithm works by finding centroids and looks at points around each centroid inside a given radius to determine which points correspond to which cluster.

Even if we try giving the KMeans algorithm the correct means of each cluster directly, it still will classify the points in the wrong way because as given, they have the wrong embedding. In that sense, each point votes for its cluster in an isolated way.

KMeans works using centroids which is the representation of the data center and each point is assigned to the nearest centroid.

(b) `adjacency_matrix = ?`

Solution:

```
y_pred = KMeans(n_clusters=3, random_state=seed).fit_predict(X)
show_data_results(X, num_plot=1)
```

(c) As given, the data points in our dataset are represented simply with their 2D Cartesian coordinates. Let's now interpret every single point as a node in a graph. Our goal is to find a way to relate every node in the graph in such way that the points that are closer together and points that are far apart maintain that relationship explicitly.

That is, we will choose to look at every point in the dataset as a vertex in a graph where the edge connection between two vertexes is determined by the weighted distances between them.

In the notebook, implement a function that takes in the input dataset and some coefficient gamma and returns the adjacency matrix A:

$$A_{i,j} = e^{-\gamma||x_i - x_j||^2} \tag{4}$$

where x_i and x_j represent each point in the provided dataset, γ is positive. You may find the `distance` module from `scipy.spatial` useful.

Is this a directed or an undirected graph?

Solution: Undirected. See solution notebook.

(d) The degree matrix of an undirected graph is a diagonal matrix which contains information about the degree of each vertex. In other words, it contains the number of edges attached to each vertex and it is given by:

$$D_{i,j} = \begin{cases} deg(v_i) & \text{if } i = j, \\ 0 & \text{otherwise.} \end{cases}$$

where the degree $\deg(v_i)$ of a vertex counts the number of times an edge terminates at that vertex. Note that in the traditional definition of the adjacency matrix, this boils down to the diagonal matrix in which element along the diagonals are column-wise sum of the adjacency matrix.

Using the same idea, **write a function that takes in the adjacency matrix as an argument and returns the degree matrix.**

Solution: See notebook solution for the function.

(e) Using γ = 7.5, **compute the adjacency matrix A, degree matrix D and the symmetrically normalized adjacency matrix matrix** M,

$$M = A^{SymNorm} = D^{-1/2}AD^{-1/2} \tag{5}$$

Solution: See solution notebook.

Note that another interpretation of the matrix M is that it shows the probability of moving/jumping from one node to another.

(f) Applying SVD decomposition on M, **write a function that selects the top 3 vectors (corresponding to the highest singular values) in the matrix U and performs the same KMeans clustering used above on them. Show the plots. What do you observe? Did it work? If so, explain why, if not, explain why not.**

Intuition: By selecting the top 3 vectors of the U matrix, we are selecting a new representation of the data points which could be seen as a construction of a low dimension embedding of the data points as mentioned in problem 3.

Solution: While it did better than the first try, it is still not separating the upper 2 clusters properly. Recall that when we write $M = U\Sigma V^T$ using the SVD decomposition for some matrix M, we are simply mapping from a set of ortho-normal vectors V_i to a different set of ortho-normal vectors U_i. That is, every U_i is scaled by σ_i and rotated in such way that orthogonality is maintained. In that sense, it also captures directions of the variances from high to low.

(g) Now let's think of the symmetrically normalized adjacency matrix obtained above as the transition matrix in of a Markov Chain. That is, it represents the probability of jumping from one node to another. In order to fully interpret M in such way, it needs to be a proper stochastic matrix which means that the sum of the elements in each column must add up to 1. **Write a function that takes in the matrix M and returns M_{stoch}, the stochastic version of M; compute the stochastic matrix.**

Solution: See solution notebook.

Using SVD decomposition on the newly obtained stochastic matrix M_{stoch}, **use your function in part (e) to select the top 3 vectors of the matrix U_{stoch} and perform the same KMeans clustering used above on them and show the plots. What do you observe? Did it work?**

Solution: See solution notebook.

It works. M_{stoch} can be viewed as the transition probabilities between the data points in a random walk on the graph. The top eigenvectors of this matrix capture the dominant patterns of the random walk, which can be used to cluster the data points. By taking the top eigenvectors, we can identify the most significant patterns of connectivity in the graph and use them to group similar data points together.

6. Homework Process and Study Group

Citing sources and collaborators are an important part of life, including being a student!
We also want to understand what resources you find helpful and how much time homework is taking, so we can change things in the future if possible.

(a) **What sources (if any) did you use as you worked through the homework?**

(b) **If you worked with someone on this homework, who did you work with?**
List names and student ID's. (In case of homework party, you can also just describe the group.)

(c) **Roughly how many total hours did you work on this homework?**

The following are old exam questions simply for your reference. You do not have to do these questions. They're coverage is redundant.

7. Machine Translation

Consider the following Machine Translation problem:

- You are learning an Encoder-Decoder model to translate sentences from Spanish into English.
- This Encoder-Decoder model consists of a single-layer RNN encoder and a single-layer RNN decoder.
- The first hidden state of the decoder is initialized with the last hidden state of the encoder.
- Words are converted into learned embeddings of length H (hidden state size), before they are passed to the model.

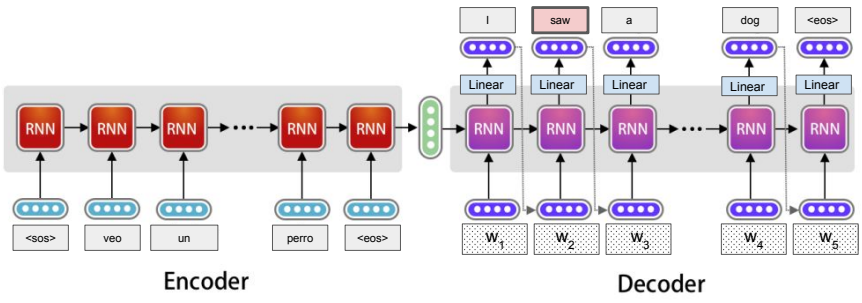


Figure 5: Translation model. The ovals with horizontal dots are learned encodings of words. The tokens $< sos >$ and $< eos >$ are "Start of Sequence" and "End of Sequence" tokens respectively. The boxes $w_1 \dots w_5$ represent the word tokens passed into the RNN decoder at each timestep (you'll fill in which tokens go here). **Solution:** Diagram modified from `http://www.adeveloperdiary.com/data-science/deep-learning/nlp/machine-translation-recurrent-neural-network-pytorch/`

(a) (4pts) Your teammate proposes stacking the encoder and decoder vertically rather than horizontally. Instead of passing the final hidden state of the encoder h_T into the decoder's first hidden state, at each timestep t, the encoder's hidden state h_t gets passed as an input to timestep t of the decoder. **State one problem with this proposed design change.**

Solution: Reasonable answers could include (a) This model does not include an easy way to handle when different-length English sentences are appropriate to translate Spanish sentences. (b) The ith English word generated can only condition on the first i Spanish words, but words in later positions may be important for translation (e.g. if the word order in the two languages is different).

(b) (3pts) In the example shown the correct translation is "I see a dog," but the translation that happened to be sampled from the model incorrectly states "I **saw** a dog".

What five tokens will be passed into the decoder during training for $w_1, w_2, \ldots, w_5$?

(HINT: Remember, during training we have access to correct supervision for translations. Don't forget that you also have special tokens $< sos >$ and $< eos >$ for the beginning and end of a sentence.)

Solution: <SOS>, I see, a, dog

During training, we pass in the ground-truth previous token.

(c) (3pts) Continuing the previous part, **what five tokens would be passed into the decoder at evaluation time for $w_1, w_2, \ldots, w_5$ when a translation is being generated?**.

(Here, you can assume that the decoder only emits a single possibility for each word.)

Solution: <SOS>, I saw, a, dog

When a translation is being generated, there is no ground-truth previous token, so we pass in the output generated by the model.

8. Self-supervised Linear Autoencoders

We consider linear models consisting of two weight matrices: an encoder $W_1 \in \mathbb{R}^{k \times m}$ and decoder $W_2 \in \mathbb{R}^{m \times k}$ (assume $1 < k < m$). The traditional autoencoder model learns a low-dimensional embedding of the n points of training data $\mathbf{X} \in \mathbb{R}^{m \times n}$ by minimizing the objective,

$$\mathcal{L}(W_1, W_2; \mathbf{X}) = \frac{1}{n} ||\mathbf{X} - W_2 W_1 \mathbf{X}||_F^2 \tag{6}$$

We will assume $\sigma_1^2 > \cdots > \sigma_k^2 > \sigma_{k+1}^2 \geq 0$ are the $k+1$ largest eigenvalues of $\frac{1}{n}\mathbf{X}\mathbf{X}^\top$. The assumption that the $\sigma_1, \ldots, \sigma_k$ are positive and distinct ensures identifiability of the principal components.

Consider an ℓ_2-regularized linear autoencoder where the objective is:

$$\mathcal{L}_\lambda(W_1, W_2; \mathbf{X}) = \tfrac{1}{n} ||\mathbf{X} - W_2 W_1 \mathbf{X}||_F^2 + \lambda \|W_1\|_F^2 + \lambda \|W_2\|_F^2. \tag{7}$$

where $\|\cdot\|_F^2$ represents the Frobenius norm squared of the matrix (i.e. sum of squares of the entries).

(a) You want to use SGD-style training in PyTorch (involving the training points one at a time) and self-supervision to find W_1 and W_2 which optimize (7) by treating the problem as a neural net being trained in a supervised fashion. **Answer the following questions and briefly explain your choice:**

(i) **How many linear layers do you need?**

- ☐ 0
- ☐ 1
- ☐ 2

☐ 3

Solution: **2**, we would use two linear layers, one for the encoder, one for the decoder.

(ii) **What is the loss function that you will be using?**

☐ `nn.L1Loss`

☐ `nn.MSELoss`

☐ `nn.CrossEntropyLoss`

Solution: We should use **MSE-Loss** to train the model (reconstruction under l2-loss) since what we want is for each vector to be close to its reconstruction in a squared-error sense.

(iii) **Which of the following would you need to optimize (7) exactly as it is written? (Select all that are needed)**

☐ Weight Decay

☐ Dropout

☐ Layer Norm

☐ Batch Norm

☐ SGD optimizer

Solution: We need to use **Weight Decay** to achieve the desired regularization and the problem states that we are doing "SGD-Style", training. Therefore we use the **SGD Optimizer**.

(b) **Do you think that the solution to (7) when we use a small nonzero λ has an inductive bias towards finding a W_2 matrix with approximately orthonormal columns? Argue why or why not?**
(Hint: Think about the SVDs of $W_1 = U_1\Sigma_1 V_1^\top$ and $W_2 = U_2\Sigma_2 V_2^\top$. You can assume that if a $k \times m$ or $m \times k$ matrix has all k of its nonzero singular values being 1, then it must have orthonormal rows or columns. Remember that the Frobenius norm squared of a matrix is just the sum of the squares of its singular values. Further think about the minimizer of $\frac{1}{\sigma^2} + \sigma^2$. Is it unique?)

Solution: If there were no regularization terms, we know that all the optimizers have to have W_2W_1 acting like a projection matrix that projects onto the k largest singular vectors of X.

This means that the W_2W_1 to minimize the main loss has to be the identity when restricted to the subspace spanned by the k largest singular vectors of X.

Therefore we would expect W_1, W_2 be approximate psuedo-inverses of each other since they are not square, and the rank of either one is at most k.

Therefore regularizing by penalizing the Frobenius norms forces us to consider:

$$\begin{aligned}\|W_1\|_F^2 + \|W_2\|_F^2 &= \|\Sigma_1\|_F^2 + \|\Sigma_2\|_F^2 \\ &= \sum_{i=1}^{k} \left(\sigma_i^2 + \frac{1}{\sigma_i^2}\right)\end{aligned}$$

where W_1 is bringing the σ_i terms and its approximate pseudo-inverse W_2 is bringing the $\frac{1}{\sigma_i}$ for its singular values.

Minimizing $\frac{1}{\sigma^2} + \sigma^2$ by taking derivatives results in setting $0 = -\frac{2}{\sigma^3} + 2\sigma$ which has a unique non-negative real solution at $\sigma = 1$, and so this is the unique minimizer since clearly this expression goes to ∞ as $\sigma \to \infty$ or $\sigma \to 0$.

If the λ is small enough, then the optimization essentially decouples: the main loss forces W_2 and W_1 to be pseudoinverses and to have the product W_2W_1 project onto the supspace spanned by the k singular vectors of X whose singular values are largest; and the regularization term forces the individual W_2 and W_1 to have all nonzero singular values as close to 1 as possible.

Once $\sigma_i \approx 1$, the matrix has approximately orthonormal columns for W_2 and approximately orthonormal rows for W_1. You can see this by simply writing $W = U\Sigma V^\top$ and then noticing for a tall W that $W^\top W = V\Sigma^\top \Sigma V^\top \approx VIV^\top = I$ and similarly for $WW^\top$ for a wide W.

Contributors:

- Dhruv Shah.
- Olivia Watkins.
- Jerome Quenum.
- Anant Sahai.
- Anrui Gu.
- Matthew Lacayo.
- Past EECS 282 and 227 Staff.
- Linyuan Gong.
- Kumar Krishna Agrawal.
- Romil Bhardwaj.
- Sheng Shen.

