Deep Learning Midterm

April 1, 2020

Total number of points is 105. Maximum score is achieved with any score between 100 and 105 points.

Tejaishwarya Gagadam, tg1779

1 LSTMs don't solve vanishing gradient [10 pts]

Background: RNNs
Consider initial state $\boldsymbol{h}_0$ and input sequence $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T)$. The RNN state $\boldsymbol{h}_t$ is a function of the previous hidden state and current input, usually

$$\boldsymbol{h}_t = \sigma(\boldsymbol{W}_1 \boldsymbol{h}_{t-1} + \boldsymbol{W}_2 \boldsymbol{x}_t)$$

Usually, loss $\mathcal{L}_t$ at time t is a function of the state $\boldsymbol{h}_t$. We minimise total loss $\mathcal{L} = \sum_t \mathcal{L}_t$. The gradient $\partial\mathcal{L}/\partial\boldsymbol{W} = \sum_t \partial\mathcal{L}_t/\partial\boldsymbol{W}$ and $\partial\mathcal{L}_\tau/\partial\boldsymbol{W}$ depends on $\partial\boldsymbol{h}_\tau/\partial\boldsymbol{h}_t$ for all $\tau > t$, where $\partial\boldsymbol{h}_\tau/\partial\boldsymbol{h}_t$ is what vanishes or explodes due to repeated application of matrix $\boldsymbol{W}_1$. We say that RNNs can't learn long dependencies when gradients vanish or explode from $\mathcal{L}_\tau$ back to $\boldsymbol{h}_t$ (and therefore to $\boldsymbol{W}_1$) for $\tau \gg t$, due to vanishing or exploding in $\partial\boldsymbol{h}_\tau/\partial\boldsymbol{h}_t$.

Background: LSTMs
The LSTM maintains an auxiliary cell state $\boldsymbol{c}$ that helps copy the hidden state $\boldsymbol{h}$ forward in time as $\boldsymbol{x}$ is processed. Letting $\boldsymbol{f}_t, \boldsymbol{i}_t, \tilde{\boldsymbol{c}}_t$ and $\boldsymbol{o}_t$ be nonlinear functions of $\boldsymbol{h_{t-1}}$ and $\boldsymbol{x}_t$ and letting $\odot$ denote element-wise product, the LSTM hidden state computation for $\boldsymbol{c}$ and $\boldsymbol{h}$ is defined by

$$\boldsymbol{c}_t = \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \tilde{\boldsymbol{c}}_t; \tag{1}$$

$$\boldsymbol{h}_t = \boldsymbol{o}_t \odot \tanh(\boldsymbol{c}_t). \tag{2}$$

In the update of the cell, the *don't forget* gate $\boldsymbol{f}_t$ determines the influence of previous cell states on the next. The *input* gate $\boldsymbol{i}_t$ determines the influence of new information on the cell. $\boldsymbol{f}_t$, $\boldsymbol{i}_t$, $\boldsymbol{o}_t$ and $\tilde{\boldsymbol{c}}_t$ are defined via usual RNN recurrences:

$$\begin{aligned}
\tilde{\boldsymbol{c}}_t &= \tanh(\boldsymbol{W}_{gh}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{gx}\boldsymbol{x}_t + \boldsymbol{b}_g);\\
\boldsymbol{f}_t &= \sigma(\boldsymbol{W}_{fh}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{fx}\boldsymbol{x}_t + \boldsymbol{b}_f);\\
\boldsymbol{i}_t &= \sigma(\boldsymbol{W}_{ih}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{ix}\boldsymbol{x}_t + \boldsymbol{b}_i);\\
\boldsymbol{o}_t &= \sigma(\boldsymbol{W}_{oh}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{ox}\boldsymbol{x}_t + \boldsymbol{b}_o).
\end{aligned}$$

Do LSTMs Still Have Vanishing Gradients?
Assume that losses $\mathcal{L}_t$ at each time step are a function of hidden state $\boldsymbol{h}_t$ (eq. 2). Consider the gradient $\partial\mathcal{L}_t/\partial\boldsymbol{W}$ for all eight matrices $\boldsymbol{W}$. Because there is a path $\boldsymbol{W} \to \boldsymbol{c} \to \boldsymbol{h} \to \mathcal{L}_t$, the cell update (eq. 1) affects the gradients $\partial\mathcal{L}_t/\partial\boldsymbol{W}$. The gradient for $\boldsymbol{W}$ is accumulated through both terms in the sum of (eq. 1).

- **Answer:** From the LSTM's auxiliary cell state definition: (eq. 1), we can see that the term $\boldsymbol{f}_t \odot \boldsymbol{c}_{t-1}$ contains the *don't forget gate*. This gate helps the system decide at every time step, if the previous state's information should be remembered or forgotten. This term is credited to solving the vanishing gradient problem in Recurrent Neural Networks.

Back-propagation through time shows:

$$\frac{\partial \mathcal{L}_t}{\partial \boldsymbol{W}} = \frac{\partial \mathcal{L}_t}{\partial \boldsymbol{h}_t}\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{c}_t}\cdots\frac{\partial \boldsymbol{c}_2}{\partial \boldsymbol{c}_1}\frac{\partial \boldsymbol{c}_1}{\partial \boldsymbol{W}}$$

$$= \frac{\partial \mathcal{L}_t}{\partial \boldsymbol{h}_t}\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{c}_t}\left[\prod_{t=2}\frac{\partial \boldsymbol{c}_t}{\partial \boldsymbol{c}_{t-1}}\right]\frac{\partial \boldsymbol{c}_1}{\partial \boldsymbol{W}}$$

The term in the square brackets, as shown in the equation above, causes the gradients to either explode or vanish. In LSTM, due to the auxiliary cell state and the hidden state, as seen in equations (eq. 1) and (eq. 2) this term in the square bracket will yield an additive gradient. Since they are additive in nature, the gradients will not explode/vanish.

$$\frac{\partial \boldsymbol{c}_t}{\partial \boldsymbol{c}_{t-1}} = \sigma^{'}\left(\boldsymbol{W}[\boldsymbol{h}_{t-1}, \boldsymbol{x}_t]\right)\boldsymbol{W}\boldsymbol{o}_{t-1} \odot \tanh^{'}(\boldsymbol{c}_{t-1})\boldsymbol{c}_{t-1}$$

$$+\boldsymbol{f}_t$$
$$+\sigma^{'}\left(\boldsymbol{W}[\boldsymbol{h}_{t-1}, \boldsymbol{x}_t]\right)\boldsymbol{W}\boldsymbol{o}_{t-1} \odot \tanh^{'}(\boldsymbol{c}_{t-1})\tilde{\boldsymbol{c}}_t$$
$$+\sigma^{'}\left(\boldsymbol{W}[\boldsymbol{h}_{t-1}, \boldsymbol{x}_t]\right)\boldsymbol{W}\boldsymbol{o}_{t-1} \odot \tanh^{'}(\boldsymbol{c}_{t-1})\boldsymbol{i}_t$$

However, if one of the four additive terms, as seen in the equation above, is more dominating than the others, the larger terms magnitude will outweigh the others. So the resultant will resemble the method that vanilla RNN takes to backpropagate through time, which means that there is still a possibility that this term might vanish/blow up!

2 Optimisation [25 pts]

Consider minimising the following loss function over θ:

$$L(\hat{\boldsymbol{Y}}_\theta(\boldsymbol{X}), \boldsymbol{y}) = \frac{1}{N}\sum_i \ell(\hat{\boldsymbol{y}}_\theta(x_i), y_i)$$

For example, it could be multi-class classification where $\hat{\boldsymbol{y}}_\theta(x_i)$ is a vector of probabilities for the output classes.

(a) **Answer:** The Gradient Descent (GD) Equation is

$$\theta_{k+1} \leftarrow \theta_k - \gamma_k \frac{\partial L(\hat{\boldsymbol{Y}}_\theta(\boldsymbol{X}), \boldsymbol{y})}{\partial \theta_k}$$

$$\theta_{k+1} \leftarrow \theta_k - \gamma_k \frac{1}{N}\frac{\partial(\sum_i \ell(\hat{\boldsymbol{y}}_\theta(x_i), y_i))}{\partial \theta_k}$$

where,

- γ_k is the step size
- θ_{k+1} is the updated value after k-th iteration
- θ_k is the updated value before k-th iteration

(b) The Stochastic Gradient Descent (SGD) Equation, assuming a mini batch of size M is:

$$\theta_{k+1} \leftarrow \theta_k - \gamma_k \frac{1}{|M_i|} \sum_{j \in M_i} \frac{\partial \ell_j(\hat{\boldsymbol{y}}_\theta(x_j), y_j)}{\partial \theta_k}$$

(c) We prefer smaller batch sizes not only because they converge sooner, but also because it results in regularization. Smaller batch sizes result in noisy steps, which can help us skip over unwanted local optimas. Since deep neural networks are easily prone to over-fitting, small batches can offer regularization and reduce the test error.

(d) A good batch size for K class classification would be slightly more than or equal to the number of classes - that is slightly more than or equal to K. This is because we want the mini batches to represent the entire dataset in each batch, so this is a way to enforce that that condition.

(e)
- Stochastic Gradient Descent (SGD) is noisy compared to Gradient Descent (GD). In the initial stages of training, the noise experienced due to SGD is small when compared to the information in the gradient, since have not accumulated gradients yet - so the steps taken in SGD are virtually comparable to GD's.
- Due to this noise produced by SGD, it can help stepping over unwanted shallow local optimas - a phenomenon known as annealing, whereas there are high chances to get stuck in a local optimas when we use GD. However, if the loss function has only one global optima, both SGD and GD will converge to this point - SGD being more computationally efficient.

(f)
- In standard GD, we update the weights based on the accumulated errors over all the samples in the training set. However, in SGD we update the weights with every training sample incrementally (starting with a randomly picked one). So SGD results in faster convergence, since it updates the weights more frequently.
- Since the path to find the global minima/maxima is quite noisy for SGD, it can gloss over local optimas. However, GD could be stuck in a local optima, and assume that it reached a global optima.

(g) Here is the "stochastic heavy ball interpretation" of the SGD with momentum formula. In the following, let $\boldsymbol{w}$ be weights, γ be learning rate, and f be loss function:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \gamma_t \nabla f_i(\boldsymbol{w}_t) + \beta_t(\boldsymbol{w}_t - \boldsymbol{w}_{t-1})$$

- The β_t term refers to the *dampening factor*. It is called a heavy ball, because the use of these additional terms $\beta_t(\boldsymbol{w}_t - \boldsymbol{w}_{t-1})$ will help stabilize the oscillatory movement seen in SGD. Similar to how a heavy ball rolls down a hill, and since this ball will have momentum it will not experience immediate changes in it's path due to external forces/changes in the space.
- The appropriate range of β_t is [0,1). If β_t was < 0, the algorithm is simply standard Gradient Descent. If β_t was > 1, the equation will blow up eventually.
- A smaller β_t value would mean a higher impact due to external changes. A large β_t value refers to lesser reaction due to these changes. In practice, it is advisable to use $\beta_t = 0.9$ or 0.99.

(h) Here are the equations for RMS Prop and Adam:

RMS Prop

$$\boldsymbol{v}_{t+1} = \alpha \boldsymbol{v}_t + (1-\alpha)\nabla f_i(\boldsymbol{w}_t)^2$$
$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \gamma \frac{\nabla f_i(\boldsymbol{w}_t)}{\sqrt{\boldsymbol{v}_{t+1}} + \epsilon}$$

- $\boldsymbol{v}_{t+1}$ is the 2^{nd} moment estimate
- γ is the learning rate
- ϵ helps avoid a 0 in the denominator (it is very small quantity, of the order 10^{-7}), which will make minimal impact.
- $\nabla f_i(\boldsymbol{w}_t)^2$ ensures that the each vector element is squared individually
- In the first equation we are calculating the exponential moving average to estimate the noisy quantity that changes over time. This adaptive method aims to up weight newer values, and assign lower weights to the values in the past - since new values give us is more information. This method does this by exponentially down-weighting the older values at every step by a constant α.
- α is a constant between 0 and 1. This works as a dampening agent, and lowers the older values till they are no longer important in the exponential moving average.
- The second equation is a modified update equation. It normalizes the gradient by Root Mean Square of the second moment (+ ϵ). If the gradient is a small value, it is equivalent to dividing the gradient by the standard deviation.
- We don't subtract the mean because the method maintains an exponential moving average for a non-central second moment.

Adam

$$\boldsymbol{m}_{t+1} = \beta \boldsymbol{m}_t + (1-\beta)\nabla f_i(\boldsymbol{w}_t)$$
$$\boldsymbol{v}_{t+1} = \alpha \boldsymbol{v}_t + (1-\alpha)\nabla f_i(\boldsymbol{w}_t)^2$$
$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \gamma \frac{\boldsymbol{m}_t}{\sqrt{\boldsymbol{v}_{t+1}} + \boldsymbol{\epsilon}}$$

(replaces numerator of RMS prop with $\boldsymbol{m}_t$)

- $\boldsymbol{m}_{t+1}$ is the momentum's exponential moving average
- $\boldsymbol{v}_{t+1}$ is the 2^{nd} moment estimate
- Adaptive Moment Estimation (Adam) is essentially RMSprop + momentum.
- In the first equation, the momentum is updated to an exponential moving average
- We choose a step-size and we don't have to change it, since it is being taken care of by β.
- The momentum is utilized to update weights instead of mini-batch gradients as seen in RMS Prop. Since the momentum values are scaled down, it has a smoother descent trajectory compared to RMS Prop.
- Adam accommodates bias correction as well (not shown in the equation above) unlike RMS Prop, hence Adam is preferred over RMS Prop for problems that have an oscillatory trajectory.
- RMS Prop takes up more computational efficiency compared to Adam. Adaptive methods (like, RMS Prop and Adam) tend to generalize poorly, compared to SGD. RMS Prop is more suitable for spare problems

(i)
```
if val_loss > min(val_losses[: - NONMONO]) AND LR_DECAY > 0:
    optimiser.param_groups[0]['lr'] /= LR_DECAY
```

- Depending on hyperparamter `NONMONO`, when the desired condition is in the vicinity, the algorithm wants to know if we over-shot the optima, or we are on the right track. It updates the learning rate accordingly, to ensure that we reach our optimal point, without missing it.
- The conditional statement checks if the current loss is greater than the minimum loss encountered until now. If it is not, the new loss would be the lowest loss yet and we maintain the same learning rate, since we are heading in the right direction.
- If the condition is met, that is, the current epoch loss is greater than the lowest loss recorded until now, the algorithm updates the learning

rate to a smaller value by dividing it with LR_DECAY. This change in the learning rate ensures that smaller steps are taken towards the optima and we don't over-shoot it. The direction is adjusted accordingly.

3 Energy based models [25 pts]

An unconditional EBM (suitable for self-supervised learning) is a scalar-valued function $F(\boldsymbol{y})$ that outputs low values if $\boldsymbol{y}$ looks like samples from the training set (i.e. is in the region of high training data density), and higher values if $\boldsymbol{y}$ is outside the region of high training data density. A conditional EBM (suitable for supervised learning and prediction-based self-supervised learning) is a function $F(\boldsymbol{x}, \boldsymbol{y})$ that takes low values if $\boldsymbol{x}$ and $\boldsymbol{y}$ are "compatible" (as defined by the training set), and higher values if they are not.

The energy function of an EBM is parameterised by a trainable parameter vector $\boldsymbol{w}$. Given a training set, a good loss functional $\mathcal{L}(\boldsymbol{w})$, when minimised, must shape the energy function so as to give lower energy to regions of high training data density, and higher energies outside.

(a) **Answer:** Given a $\boldsymbol{x}$, we can infer a compatible $\boldsymbol{y}$ using the energy function: $F(\boldsymbol{x}, \boldsymbol{y})$. $F(\boldsymbol{x}, \boldsymbol{y})$ determines the level of dependency between (x,y) pairs. That is an input $\boldsymbol{x}$ produces the most compatible $\hat{\boldsymbol{y}}$, selected from a set of points y, where the value of $F(\boldsymbol{x}, \boldsymbol{y})$ is lowest.

$$\hat{\boldsymbol{y}} = argmin_y F(\boldsymbol{x}, \boldsymbol{y}) \tag{3}$$

(b) Give the expression for $\mathbb{P}(\boldsymbol{y} \mid \boldsymbol{x})$ assuming a Gibbs-Boltzmann distribution. [3 pts]

$$\mathbb{P}(\boldsymbol{y} \mid \boldsymbol{x}) = \frac{\exp(-\beta F(\boldsymbol{x}, \boldsymbol{y}))}{\int_y \exp\left(-\beta F(\boldsymbol{x}, y)\right)} \tag{4}$$

where β is a positive constant.

(c) The deterministic and probabilistic expressions for the free energy $F(\boldsymbol{x}, \boldsymbol{y})$ as a function of $E(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$ are given by:

When the model depends on a latent variable $\boldsymbol{z}$:

$$\mathbb{P}(\boldsymbol{y}, \boldsymbol{z} \mid \boldsymbol{x}) = \frac{\exp(-\beta F(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}))}{\int_y \exp\left(-\beta F(\boldsymbol{x}, y, \boldsymbol{z})\right)}$$

When we marginalize:

$$\mathbb{P}(\boldsymbol{y} \mid \boldsymbol{x}) = \int_{\vec{z}} \mathbb{P}(\boldsymbol{y}, \boldsymbol{z} \mid \boldsymbol{x})$$

$$\mathbb{P}(\boldsymbol{y} \mid \boldsymbol{x}) = \frac{\int_y \exp(-\beta E(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}))}{\int_y \int_{\boldsymbol{z}} \exp(-\beta E(\boldsymbol{x}, y, \boldsymbol{z}))}$$

$$= \frac{\exp(-\beta(-\frac{1}{\beta} \log \int_{\boldsymbol{z}} \exp(-\beta E(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}))))}{\int_y \exp(-\beta E(\boldsymbol{x}, y, \boldsymbol{z}))}$$

$$= \frac{\exp(-\beta F_\beta(\boldsymbol{x}, \boldsymbol{y})))}{\int_y \exp(-\beta F_\beta(\boldsymbol{x}, y))}$$

$F_\beta(\boldsymbol{x}, y)$ is the Free Energy

$$F_\beta(\boldsymbol{x}, y) = -\frac{1}{\beta} \log \int_{\boldsymbol{z}} \exp(-\beta E(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}))$$

(d) The expression of the negative log-likelihood loss, which is designed to maximise the likelihood that our model assigns to the $\boldsymbol{y}^{(i)}$ given the $\boldsymbol{x}^{(i)}$ in the training set: $\prod_i \mathbb{P}(\boldsymbol{y}^{(i)} \mid \boldsymbol{x}^{(i)})$ cab be derived as follows:

$$\mathbb{P}(\boldsymbol{y}^1, ..., \boldsymbol{y}^N \mid \boldsymbol{x}^1, ..., \boldsymbol{x}^N, \boldsymbol{w}) = \prod_i \mathbb{P}(\boldsymbol{y}^{(i)} \mid \boldsymbol{x}^{(i)}, \boldsymbol{w})$$

using maximum likelihood we try to find the values of the trainable parameter vector $\boldsymbol{w}$ which will maximize this product (or which minimizes the negative log of this product)

$$-\log(\mathbb{P}(\boldsymbol{y}^{(i)} \mid \boldsymbol{x}^{(i)}, \boldsymbol{w})) = \sum_i^N -\log(\mathbb{P}(\boldsymbol{y}^{(i)} \mid \boldsymbol{x}^{(i)}))$$

$$-\log \mathbb{P}(\boldsymbol{y}^{(i)} \mid \boldsymbol{x}^{(i)}, \boldsymbol{w}) = \sum_i^N \beta F_{\boldsymbol{w}}(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}) + \log \int_y \exp(-\beta F_{\boldsymbol{w}}(\boldsymbol{x}^{(i)}, y))$$

$$\mathcal{L}(\boldsymbol{w}) = \frac{1}{N} \sum_i^N \left[F_{\boldsymbol{w}}(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}) + \frac{1}{\beta} \log \int_y \exp(-\beta F_{\boldsymbol{w}}(\boldsymbol{x}^{(i)}, y)) \right]$$

$$\frac{\partial \mathcal{L}(\boldsymbol{w})}{\partial \boldsymbol{w}} = \frac{\partial F_{\boldsymbol{w}}(\boldsymbol{x}^{(i)}, \boldsymbol{y}^i)}{\partial \boldsymbol{w}} - \int_y \frac{\partial F_{\boldsymbol{w}}(\boldsymbol{x}^{(i)}, y)}{\partial \boldsymbol{w}} \mathbb{P}(\boldsymbol{y}^{(i)} \mid \boldsymbol{x}^{(i)})$$

$$\frac{\partial \mathcal{L}(\boldsymbol{w})}{\partial \boldsymbol{w}} = F_{\boldsymbol{w}}(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}) + F_{\beta, \boldsymbol{w}}(\boldsymbol{x}^{(i)}, y)$$

where $\mathbb{P}(\boldsymbol{y}^{(i)} \mid \boldsymbol{x}^{(i)})$ is the expression from Gibbs-Boltzmann distribution (shown above).

(e) **Noise Contrastive Estimator's** objective function is a contrastive method, which tries to maximize the cost function without any contraints on normalization which allows us to operate using un-normalized functions.

$S(u, v)$: is a similarity metric between two feature maps/vectors/Convolution Net outputs as a measure of their cosine similarity. $f(u), g(u)$ can be

thought of as independent layers on top of convolutional feature extractors/layers.

Using mini-batches, each batch will consist of one similar/positive pair, and many negative/dissimilar pairs. Using these we can measure the similarity between the transformed feature vector $\boldsymbol{I_t}$ and the other feature vectors in the mini-batch, one positive and the rest negative pairs. Then we compute the score of a softmax-like function on the positive pair. So, maximizing a softmax score means that we minimize the rest of the scores. Which aligns with what we want from our Energy-Based Model. The final loss function, as shown below, pushes the energy down on similar/positive pair locations, and pushes up on dissimilar/negative pair locations.

$$h(\boldsymbol{v_I}, \boldsymbol{v_{I^t}}) = \frac{\exp\left(\frac{s(\boldsymbol{v_I}, \boldsymbol{v_{I^t}})}{\tau}\right)}{\exp\left(\frac{s(\boldsymbol{v_I}, \boldsymbol{v_{I^t}})}{\tau}\right) + \sum_{I' \in D_N} \exp\left(\frac{s(\boldsymbol{v_I}, \boldsymbol{v_{I'}})}{\tau}\right)}$$

$$L_{NCE}(\boldsymbol{I}, \boldsymbol{I^t}) = -\log\left[h(f(\boldsymbol{v_I}), g(\boldsymbol{v_{I^t}})\right] - \sum_{I' \in D_N} \log\left[1 - h(g(\boldsymbol{v_{I^t}}), f(\boldsymbol{v_{I'}}))\right]$$

(f) Another class of Energy-Based Model training is called Architectural methods. These methods are different ways of limiting the information capacity.

For example,

- **K-Means Clustering**:
  - This is an unsupervised learning algorithm where we cluster the data in various groups (k clusters) based on their nearest means (of the cluster centroids).
  - K-means uses a energy-based model with the energy function $F(\boldsymbol{x}, \boldsymbol{y}) = ||y - Wz||^2$, where z is a 1-hot vector. With the values of y and k, we can infer the k possible columns of W which minimizes the reconstruction error/energy function.
  - This method ensures that the volume of the low energy regions are bounded by the number of prototypes k.
  - To train the algorithm, we use an architectural method where we use latent variable z, to choose the column of W closest to y and then try to move even closer by taking a gradient step and repeat the process.
  - We can observe how we don't intend to push the energy levels up anywhere (like we do in contrastive methods), but we try to push down the energy values in certain regions.
- **Variational Auto Encoders**:
  - VAEs are generative models which have well defined latent spaces.

  - This Architectural method builds an energy function $F(\boldsymbol{x}, \boldsymbol{y})$ which have low energy regions by the means of regularization.
  - For VAE, the loss function is defined as: $l(\boldsymbol{x}, \hat{\boldsymbol{x}}) = l_{reconstructionerror} + \beta l_{KL}(\boldsymbol{z}, N(0, I_d))$.
  - The regularization term is on the latent layer, which enforces a bottleneck of information using some specific Gaussian structure on the latent space. We use a penalty term $l_{KL}(\boldsymbol{z}, N(0, I_d))$. Without this regularization term, VAE will resemble an autoencoder, which usually leads to over-fitting and won't exhibit the generative properties we need.

- **Sparse Coding:**
  - Sparse coding can be defined as an unconditional regularized latent-variable energy based model. It approximates the data with a piecewise linear function.
  - The energy function can be defined as: $F(\boldsymbol{x}, \boldsymbol{y}) = ||y - Wz||^2 + \lambda||z||_{L1}$.
  - This Architectural model also tries to lower the energy values in certain regions (determined by the latent variable z), by using a regularization term.