Self-Explainable / Concept-Based Models

Stefano Teso

Advanced Topics in Machine Learning and Optimization 2022-23

White-box models

Examples: shallow DTs, sparse linear models, and rules lists over interpretable input variables.

Pros

- Many well-known, easy to use models.
- Make explanations available for free.
- High performance on certain types of tabular data.

Cons

- No support for representation learning.
- Low performance on non-tabular data, e.g., text, images, audio, video.

Black-box models

Examples: neural networks, non-linear kernel machines, ensamble methods.

Pros

- Many well-known, easy to use examples.
- High performance on non-tabular data like images and text.

Cons

- Support representation learning.
- Opaque, explanations must be obtained in a post-hoc fashion.
- Post-hoc attribution techniques can be contradictory or ambiguous (e.g., input gradients) or expensive to compute and have high variance (e.g., LIME).

Extra Pro: post-hoc explainers do not require to modify or retrain the model, e.g., download a pre-trained model and compute explanations for it. **Is this always desirable/necessary?**

Black-box models

Examples: neural networks, non-linear kernel machines, ensamble methods.

Pros

- Many well-known, easy to use examples.
- High performance on non-tabular data like images and text.

Cons

- Support representation learning.
- Opaque, explanations must be obtained in a post-hoc fashion.
- Post-hoc attribution techniques can be contradictory or ambiguous (e.g., input gradients) or expensive to compute and have high variance (e.g., LIME).

Extra Pro: post-hoc explainers do not require to modify or retrain the model, e.g., download a pre-trained model and compute explanations for it. Is this always desirable/necessary?

Black-box models

Examples: neural networks, non-linear kernel machines, ensamble methods.

Pros

- Many well-known, easy to use examples.
- High performance on non-tabular data like images and text.

Cons

- Support representation learning.
- Opaque, explanations must be obtained in a post-hoc fashion.
- Post-hoc attribution techniques can be contradictory or ambiguous (e.g., input gradients) or expensive to compute and have high variance (e.g., LIME).

Extra Pro: post-hoc explainers do not require to modify or retrain the model, e.g., download a pre-trained model and compute explanations for it. **Is this always desirable/necessary?**

Tree Regularization

- Regularizing black-boxes to be more transparent is a well-known strategy
- Idea: take a black-box f_θ and fine-tune it to make it more interpretable.

Example

If f_θ is a dense linear model, add a sparsifying L_1 regularizer so that its weight vector contains many zeros.

This makes the model more *simulatable* [Lipton, 2018]: "take in input data together with the parameters of the model and in reasonable time step through every calculation required to produce a prediction".

- Can we go **generalize** this idea? Can we make a neural network **behave like a decision tree** – so as to facilitate conversion into one using, e.g., LIME?

- Regularizing black-boxes to be more transparent is a well-known strategy
- Idea: take a black-box f_θ and fine-tune it to make it more interpretable.

Example

If f_θ is a dense linear model, add a sparsifying L_1 regularizer so that its weight vector contains many zeros.

This makes the model more *simulatable* [Lipton, 2018]: "take in input data together with the parameters of the model and in reasonable time step through every calculation required to produce a prediction".

- Can we go **generalize** this idea? Can we make a neural network **behave like a decision tree** – so as to facilitate conversion into one using, e.g., LIME?

- Regularizing black-boxes to be more transparent is a well-known strategy
- Idea: take a black-box f_θ and fine-tune it to make it more interpretable.

Example

If f_θ is a dense linear model, add a sparsifying L_1 regularizer so that its weight vector contains many zeros.

This makes the model more *simulatable* [Lipton, 2018]: "take in input data together with the parameters of the model and in reasonable time step through every calculation required to produce a prediction".

- Can we go **generalize** this idea? Can we make a neural network **behave like a decision tree** – so as to facilitate conversion into one using, e.g., LIME?

- Regularizing black-boxes to be more transparent is a well-known strategy
- Idea: take a black-box f_θ and fine-tune it to make it more interpretable.

Example

If f_θ is a dense linear model, add a sparsifying L_1 regularizer so that its weight vector contains many zeros.

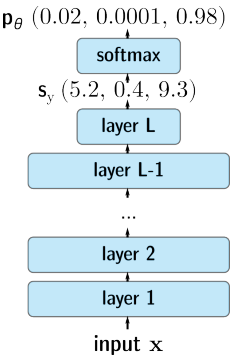
This makes the model more *simulatable* [Lipton, 2018]: "take in input data together with the parameters of the model and in reasonable time step through every calculation required to produce a prediction".

- Can we go **generalize** this idea? Can we make a neural network **behave like a decision tree** – so as to facilitate conversion into one using, e.g., LIME?

Tree Regularization [Wu et al., 2018]

- Take a regular neural network $p_\theta(y \mid \mathbf{x})$ and a training set $S = \{(\mathbf{x}_i, y_i) : i = 1, \ldots, m\}$. Normally, you would train it by minimizing the following empirical loss:

$$\frac{1}{|S|} \sum_{(\mathbf{x},y)\in S} -\log p_\theta(\mathbf{x}, y)$$



The structure of a typical feed-forward neural network with L layers.

Tree Regularization [Wu et al., 2018]

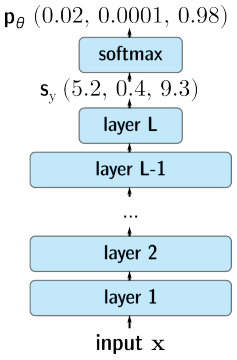
- Take a regular neural network $p_\theta(y \mid \mathbf{x})$ and a training set $S = \{(\mathbf{x}_i, y_i) : i = 1, \ldots, m\}$. Normally, you would train it by minimizing the following empirical loss:

$$\frac{1}{|S|} \sum_{(\mathbf{x},y)\in S} -\log p_\theta(\mathbf{x}, y)$$

- Minimize the following **augmented loss** instead:

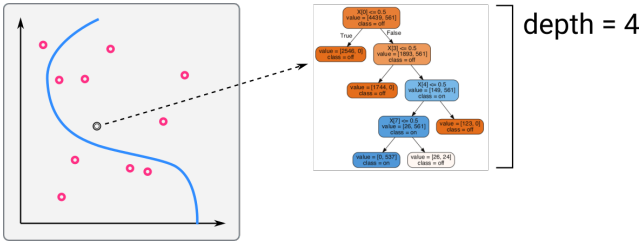
$$\frac{1}{|S|} \sum_{(\mathbf{x},y)\in S} \big(-\log p_\theta(\mathbf{x}, y) + \lambda \cdot \Omega(\theta) \big)$$

where $\Omega(\theta)$ is the **average depth** of a decision tree that fits f_θ. In other words, Ω is small only if $f_\theta(\mathbf{x})$ can be simulated – on average – by a small decision tree.



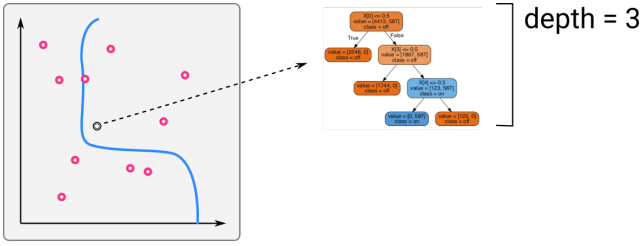
The structure of a typical feed-forward neural network with L layers.

Illustration



The tree complexity is computed at a the **black** point $\mathbf{x}$

Illustration



The tree complexity is computed at a the **black** point $\mathbf{x}$

Issues

- How to **compute** $\Omega(\theta)$?
- How to make **minimize** $\Omega(\theta)$?
- **Idea**: learn an **auxiliary regressor** $\widehat{\Omega}_\mu(\theta)$ that predicts the average depth of a decision tree that fits f_θ from the parameters θ themselves.

Issues

- How to **compute** $\Omega(\theta)$?
- How to make **minimize** $\Omega(\theta)$?
- **Idea**: learn an **auxiliary regressor** $\widehat{\Omega}_\mu(\theta)$ that predicts the average depth of a decision tree that fits f_θ from the parameters θ themselves.

Tree Regularization Algorithm

- Sample reference networks $\Theta = \{\theta_1, \ldots, \theta_q\}$, e.g., pre-trained models or at random.
- Sample reference instances $Q = \{\mathbf{x}_1, \ldots, \mathbf{x}_q\}$, either from the training set S or at random
- **Compute $\Omega(\theta)$** for each reference network θ_i: fit a decision tree for θ_i on the reference instances Q labeled using f_{θ_i} using `scikit-learn`, the evaluate the average tree depth over Q.
- Fit $\widehat{\Omega}_\mu(\theta)$ so that it approximates Ω on the reference models in Θ:

$$\operatorname*{argmin}_{\mu} \; (\widehat{\Omega}_\mu(\theta) - \Omega(\theta))^2 + \|\mu\|^2$$

- Fit f_θ on training set (cold start)
- Repeat:
  - Fit θ for one epoch using the modified loss:

$$\operatorname*{argmin}_{\theta} \; \frac{1}{|S|} \sum_{(\mathbf{x},y)\in S} -\log p_\theta(\mathbf{x}, y) + \lambda\, \widehat{\Omega}_\mu(\theta)$$

    The loss is now **fully differentiable**.
  - Add θ to the reference networks and fine-tune $\widehat{\Omega}_\mu$.

Tree Regularization Algorithm

- Sample reference networks $\Theta = \{\theta_1, \ldots, \theta_q\}$, e.g., pre-trained models or at random.
- Sample reference instances $Q = \{\mathbf{x}_1, \ldots, \mathbf{x}_q\}$, either from the training set S or at random
- **Compute** $\Omega(\theta)$ for each reference network θ_i: fit a decision tree for θ_i on the reference instances Q labeled using f_{θ_i} using `scikit-learn`, the evaluate the average tree depth over Q.
- Fit $\widehat{\Omega}_\mu(\theta)$ so that it approximates Ω on the reference models in Θ:

$$\operatorname*{argmin}_{\mu} \; (\widehat{\Omega}_\mu(\theta) - \Omega(\theta))^2 + \|\mu\|^2$$

- Fit f_θ on training set (cold start)
- Repeat:
  - Fit θ for one epoch using the modified loss:

$$\operatorname*{argmin}_{\theta} \; \frac{1}{|S|} \sum_{(\mathbf{x},y)\in S} -\log p_\theta(\mathbf{x}, y) + \lambda \, \widehat{\Omega}_\mu(\theta)$$

    The loss is now **fully differentiable**.
  - Add θ to the reference networks and fine-tune $\widehat{\Omega}_\mu$.

Tree Regularization Algorithm

- Sample reference networks $\Theta = \{\theta_1, \ldots, \theta_q\}$, e.g., pre-trained models or at random.
- Sample reference instances $Q = \{\mathbf{x}_1, \ldots, \mathbf{x}_q\}$, either from the training set S or at random
- **Compute** $\Omega(\theta)$ for each reference network θ_i: fit a decision tree for θ_i on the reference instances Q labeled using f_{θ_i} using `scikit-learn`, the evaluate the average tree depth over Q.
- Fit $\widehat{\Omega}_\mu(\theta)$ so that it approximates Ω on the reference models in Θ:

$$\operatorname*{argmin}_{\mu} \; (\widehat{\Omega}_\mu(\theta) - \Omega(\theta))^2 + \|\mu\|^2$$

- Fit f_θ on training set (cold start)
- Repeat:
  - Fit θ for one epoch using the modified loss:

$$\operatorname*{argmin}_{\theta} \; \frac{1}{|S|} \sum_{(\mathbf{x},y)\in S} -\log p_\theta(\mathbf{x}, y) + \lambda \, \widehat{\Omega}_\mu(\theta)$$

    The loss is now **fully differentiable**.
  - Add θ to the reference networks and fine-tune $\widehat{\Omega}_\mu$.

Tree Regularization Algorithm

- Sample reference networks $\Theta = \{\theta_1, \ldots, \theta_q\}$, e.g., pre-trained models or at random.
- Sample reference instances $Q = \{\mathbf{x}_1, \ldots, \mathbf{x}_q\}$, either from the training set S or at random
- **Compute** $\Omega(\theta)$ for each reference network θ_i: fit a decision tree for θ_i on the reference instances Q labeled using f_{θ_i} using `scikit-learn`, the evaluate the average tree depth over Q.
- Fit $\widehat{\Omega}_\mu(\theta)$ so that it approximates Ω on the reference models in Θ:

$$\operatorname*{argmin}_{\mu} \; (\widehat{\Omega}_\mu(\theta) - \Omega(\theta))^2 + \|\mu\|^2$$

- Fit f_θ on training set (cold start)
- Repeat:
  - Fit θ for one epoch using the modified loss:

$$\operatorname*{argmin}_{\theta} \; \frac{1}{|S|} \sum_{(\mathbf{x},y)\in S} -\log p_\theta(\mathbf{x}, y) + \lambda \, \widehat{\Omega}_\mu(\theta)$$

    The loss is now **fully differentiable**.
  - Add θ to the reference networks and fine-tune $\widehat{\Omega}_\mu$.

Tree Regularization Algorithm

- Sample reference networks $\Theta = \{\theta_1, \ldots, \theta_q\}$, e.g., pre-trained models or at random.
- Sample reference instances $Q = \{\mathbf{x}_1, \ldots, \mathbf{x}_q\}$, either from the training set S or at random
- **Compute** $\Omega(\theta)$ for each reference network θ_i: fit a decision tree for θ_i on the reference instances Q labeled using f_{θ_i} using `scikit-learn`, the evaluate the average tree depth over Q.
- Fit $\widehat{\Omega}_\mu(\theta)$ so that it approximates Ω on the reference models in Θ:

$$\operatorname*{argmin}_{\mu} \; (\widehat{\Omega}_\mu(\theta) - \Omega(\theta))^2 + \|\mu\|^2$$

- Fit f_θ on training set (cold start)
- Repeat:
  - Fit θ for one epoch using the modified loss:

$$\operatorname*{argmin}_{\theta} \; \frac{1}{|S|} \sum_{(\mathbf{x},y)\in S} -\log p_\theta(\mathbf{x}, y) + \lambda\, \widehat{\Omega}_\mu(\theta)$$

    The loss is now **fully differentiable**.
  - Add θ to the reference networks and fine-tune $\widehat{\Omega}_\mu$.

Tree Regularization Algorithm

- Sample reference networks $\Theta = \{\theta_1, \ldots, \theta_q\}$, e.g., pre-trained models or at random.
- Sample reference instances $Q = \{\mathbf{x}_1, \ldots, \mathbf{x}_q\}$, either from the training set S or at random
- **Compute $\Omega(\theta)$** for each reference network θ_i: fit a decision tree for θ_i on the reference instances Q labeled using f_{θ_i} using `scikit-learn`, the evaluate the average tree depth over Q.
- Fit $\widehat{\Omega}_\mu(\theta)$ so that it approximates Ω on the reference models in Θ:

$$\underset{\mu}{\text{argmin}} \; (\widehat{\Omega}_\mu(\theta) - \Omega(\theta))^2 + \|\mu\|^2$$

- Fit f_θ on training set (cold start)
- Repeat:
  - Fit θ for one epoch using the modified loss:

$$\underset{\theta}{\text{argmin}} \; \frac{1}{|S|} \sum_{(\mathbf{x},y)\in S} - \log p_\theta(\mathbf{x}, y) + \lambda \, \widehat{\Omega}_\mu(\theta)$$

    The loss is now **fully differentiable**.
  - Add θ to the reference networks and fine-tune $\widehat{\Omega}_\mu$.

Tree Regularization Algorithm

- Sample reference networks $\Theta = \{\theta_1, \ldots, \theta_q\}$, e.g., pre-trained models or at random.
- Sample reference instances $Q = \{\mathbf{x}_1, \ldots, \mathbf{x}_q\}$, either from the training set S or at random
- **Compute $\Omega(\theta)$** for each reference network θ_i: fit a decision tree for θ_i on the reference instances Q labeled using f_{θ_i} using `scikit-learn`, the evaluate the average tree depth over Q.
- Fit $\widehat{\Omega}_\mu(\theta)$ so that it approximates Ω on the reference models in Θ:

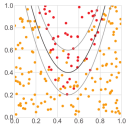
$$\underset{\mu}{\text{argmin}} \; (\widehat{\Omega}_\mu(\theta) - \Omega(\theta))^2 + \|\mu\|^2$$

- Fit f_θ on training set (cold start)
- Repeat:
  - Fit θ for one epoch using the modified loss:

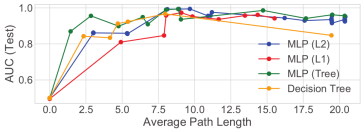
$$\underset{\theta}{\text{argmin}} \; \frac{1}{|S|} \sum_{(\mathbf{x},y)\in S} - \log p_\theta(\mathbf{x}, y) + \lambda \, \widehat{\Omega}_\mu(\theta)$$

    The loss is now **fully differentiable**.
  - Add θ to the reference networks and fine-tune $\widehat{\Omega}_\mu$.

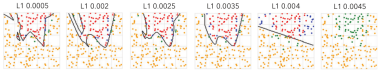
Example: Fitting a Parabola



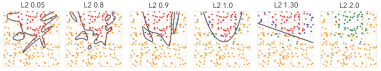
(a) Training Data and Binary Class Labels for 2D Parabola



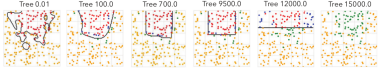
(b) Prediction quality and complexity as reg. strength λ varies



(c) Decision Boundaries with L1 regularization



(d) Decision Boundaries with L2 regularization



(e) Decision Boundaries Tree regularization

Figure 2: *2D Parabola task*: *(a)* Each training data point in 2D space, overlaid with true parabolic class boundary. *(b)*: Each method's prediction quality (AUC) and complexity (path length) metrics, across range of regularization strength λ. In the small path length regime between 0 and 5, tree regularization produces models with higher AUC than L1 or L2. *(c-e)*: Decision boundaries (black lines) have qualitatively different shapes for different regularization schemes, as regularization strength λ increases. We color predictions as true positive (red), true negative (yellow), false negative (green), and false positive (blue).

- For $\lambda = 9500$ (the exact value is not important) the tree-regularized network recovers exactly the shape of a **DT with depth** 2. Increasing λ further further flattens the tree to depth 1, at the cost of accuracy.

- Tree-based regularization has some limitations:
  - Training is either **computationally expensive**: must train $\widehat{\Omega}_\mu$ in every epoch!
  - The regularizer is **approximate**: no guarantee that $\widehat{\Omega}_\mu$ performs well, the depth prediction task is quite challenging!
  - Decision trees only make sense for **tabular data**, so the overall network is restricted to this case too.
  - Conflicts with **representation learning**

- Can we **combine** the benefits of black-box and white-box models in a more direct and efficient manner?

Yes: change the nets' architecture

- Tree-based regularization has some limitations:
  - Training is either **computationally expensive**: must train $\widehat{\Omega}_\mu$ in every epoch!
  - The regularizer is **approximate**: no guarantee that $\widehat{\Omega}_\mu$ performs well, the depth prediction task is quite challenging!
  - Decision trees only make sense for **tabular data**, so the overall network is restricted to this case too.
  - Conflicts with **representation learning**

- Can we **combine** the benefits of black-box and white-box models in a more direct and efficient manner?

Yes: change the nets' architecture

- Tree-based regularization has some limitations:
  - Training is either **computationally expensive**: must train $\widehat{\Omega}_\mu$ in every epoch!
  - The regularizer is **approximate**: no guarantee that $\widehat{\Omega}_\mu$ performs well, the depth prediction task is quite challenging!
  - Decision trees only make sense for **tabular data**, so the overall network is restricted to this case too.
  - Conflicts with **representation learning**

- Can we **combine** the benefits of black-box and white-box models in a more direct and efficient manner?

Yes: change the nets' architecture

- Tree-based regularization has some limitations:
  - Training is either **computationally expensive**: must train $\widehat{\Omega}_\mu$ in every epoch!
  - The regularizer is **approximate**: no guarantee that $\widehat{\Omega}_\mu$ performs well, the depth prediction task is quite challenging!
  - Decision trees only make sense for **tabular data**, so the overall network is restricted to this case too.
  - Conflicts with **representation learning**

- Can we **combine** the benefits of black-box and white-box models in a more direct and efficient manner?

Yes: change the nets' architecture

- Tree-based regularization has some limitations:
  - Training is either **computationally expensive**: must train $\widehat{\Omega}_\mu$ in every epoch!
  - The regularizer is **approximate**: no guarantee that $\widehat{\Omega}_\mu$ performs well, the depth prediction task is quite challenging!
  - Decision trees only make sense for **tabular data**, so the overall network is restricted to this case too.
  - Conflicts with **representation learning**

- Can we **combine** the benefits of black-box and white-box models in a more direct and efficient manner?

Yes: change the nets' architecture

- Tree-based regularization has some limitations:
  - Training is either **computationally expensive**: must train $\widehat{\Omega}_\mu$ in every epoch!
  - The regularizer is **approximate**: no guarantee that $\widehat{\Omega}_\mu$ performs well, the depth prediction task is quite challenging!
  - Decision trees only make sense for **tabular data**, so the overall network is restricted to this case too.
  - Conflicts with **representation learning**

- Can we **combine** the benefits of black-box and white-box models in a more direct and efficient manner?

Yes: change the nets' architecture

Concept-based Model (CBM)

A model f_θ is **gray-box** if it combines uninterpretable black-box components with a white-box skeleton and:

- It **automatically outputs explanations** for all of its decisions
- Its explanations are **cheap** to compute
- Its explanations are **faithful** (and hence **low-variance**)
- Features **large capacity** and **representation learning**

aka "partially interpretable models" because only parts of their decision process are transparent.

We will see different classes of CBMs:

- Self-explainable Neural Networks (SENNs) [Alvarez-Melis and Jaakkola, 2018]
- Prototypical Nets (ProtoNets) [Snell et al., 2017]
- Prototype Classification Networks (PCNs) [Li et al., 2018]
- Part-Prototype Networks (PPNets) [Chen et al., 2019]

and discuss their **promise** and **issues**

Concept-based Model (CBM)

A model f_θ is **gray-box** if it combines uninterpretable black-box components with a white-box skeleton and:

- It **automatically outputs explanations** for all of its decisions
- Its explanations are **cheap** to compute
- Its explanations are **faithful** (and hence **low-variance**)
- Features **large capacity** and **representation learning**

aka "partially interpretable models" because only parts of their decision process are transparent.

We will see different classes of CBMs:

- Self-explainable Neural Networks (SENNs) [Alvarez-Melis and Jaakkola, 2018]
- Prototypical Nets (ProtoNets) [Snell et al., 2017]
- Prototype Classification Networks (PCNs) [Li et al., 2018]
- Part-Prototype Networks (PPNets) [Chen et al., 2019]

and discuss their **promise** and **issues**

Self-explainable Neural Networks

A **linear model** has the form:

$$f(\mathbf{x}) = \text{sign}\Big(\underbrace{\sum_{i \in [d]} w_i x_i}_{\text{“score” of } \mathbf{x}} + b \Big)$$

A linear model is **sparse** if $\mathbf{w} \in \mathbb{R}^d$ few non-zero entries [Tibshirani, 1996, Ustun and Rudin, 2016] and dense otherwise. We will briefly *forget* about sparsity for now.

It is **easy** to gather an intuitive understanding of what the model does:

- $w_i > 0 \implies x_i$ correlates with, aka “votes for”, the positive class
- $w_i < 0 \implies x_i$ anti-correlates with, aka “votes against”, the positive class
- $w_i \approx 0 \implies x_i$ is irrelevant: changing it does not affect the outcome

Example: Papayas

Does a **papaya** $\mathbf{x}$ taste good?

Consider a linear classifier:

$$
\begin{aligned}
f(\mathbf{x}) = \operatorname{sign}\big(\ & 1.3 \cdot \mathbb{1}(\mathbf{x} \text{ pulp is orange}) + \\
& 0.7 \cdot \mathbb{1}(\mathbf{x} \text{ skin is yellow}) + \\
& \mathbf{0} \cdot \mathbb{1}(\mathbf{x} \text{ is round}) + \\
& -0.5 \cdot \mathbb{1}(\mathbf{x} \text{ skin is green}) + \\
& -2.3 \cdot \mathbb{1}(\mathbf{x} \text{ is moldy})\big)
\end{aligned}
$$

Figure 1: A bunch of papaya fruits.

It is easy to read off what attributes are "for" and "against" $\mathbf{x}$ being tasty **for the model** – specifically because the model encodes independence assumptions, e.g., that the shape of $\mathbf{x}$ is unrelated to its color.[1]

[1] When **explaining** a decision made by the model, **it is irrelevant whether these assumptions match how reality works**: we are explaining the model's reasoning process, or equivalently its interpretation of how reality works, not reality itself!

- **Linear models** only work for linear data and cannot perform representation learning: their only parameters are weights, and these are applied to the inputs directly!

- We already know that to turn a linear model work in a non-linear one it is sufficient to embed all points, giving:

$$p(1 \mid \mathbf{x}) = \sigma\Big(\sum_i w_i x_i\Big) \quad \mapsto \quad p(1 \mid \mathbf{x}) = \sigma\Big(\sum_i w_i \phi_i(\mathbf{x})\Big)$$

where, e.g., $\mathbf{x}$ are words in a document and $\phi(\mathbf{x})$ is a BERT or TF-IDF embedding. However, doing so **forfeits interpretability**!

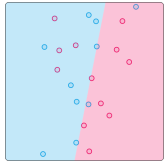


Illustration of a linear model. It cannot separate data with a complex, non-linear distribution.

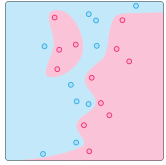


Illustration of a non-linear model. It works well with non-linear data like text, images, etc.

- **Linear models** only work for linear data and cannot perform representation learning: their only parameters are weights, and these are applied to the inputs directly!

- We already know that to turn a linear model work in a non-linear one it is sufficient to embed all points, giving:

$$p(1 \mid \mathbf{x}) = \sigma\Big(\sum_i w_i x_i\Big) \quad \mapsto \quad p(1 \mid \mathbf{x}) = \sigma\Big(\sum_i w_i \phi_i(\mathbf{x})\Big)$$

where, e.g., $\mathbf{x}$ are words in a document and $\phi(\mathbf{x})$ is a BERT or TF-IDF embedding. However, doing so **forfeits interpretability**!

- **Self-explainable neural networks** (SENNs) [Alvarez-Melis and Jaakkola, 2018], generalize linear models to **non-linear data** and **representation learning**.

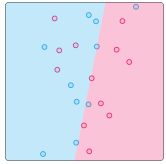


Illustration of a linear model. It cannot separate data with a complex, non-linear distribution.

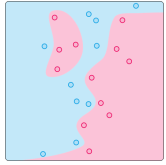


Illustration of a non-linear model. It works well with non-linear data like text, images, etc.

■ **Linear models** only work for linear data and cannot perform representation learning: their only parameters are weights, and these are applied to the inputs directly!

■ We already know that to turn a linear model work in a non-linear one it is sufficient to embed all points, giving:

$$p(1 \mid \mathbf{x}) = \sigma\Big(\sum_i w_i x_i\Big) \quad \mapsto \quad p(1 \mid \mathbf{x}) = \sigma\Big(\sum_i w_i \phi_i(\mathbf{x})\Big)$$

where, e.g., $\mathbf{x}$ are words in a document and $\phi(\mathbf{x})$ is a BERT or TF-IDF embedding. However, doing so **forfeits interpretability**!

■ **Self-explainable neural networks** (SENNs) [Alvarez-Melis and Jaakkola, 2018], generalize linear models to **non-linear data** and **representation learning**.

■ Idea: take a non-linear model (e.g., a neural net) but ensure that it **behaves like a linear model** at any given point $\mathbf{x} \in \mathbb{R}^d$!

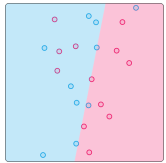


Illustration of a linear model. It cannot separate data with a complex, non-linear distribution.

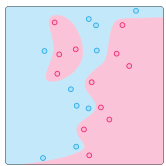
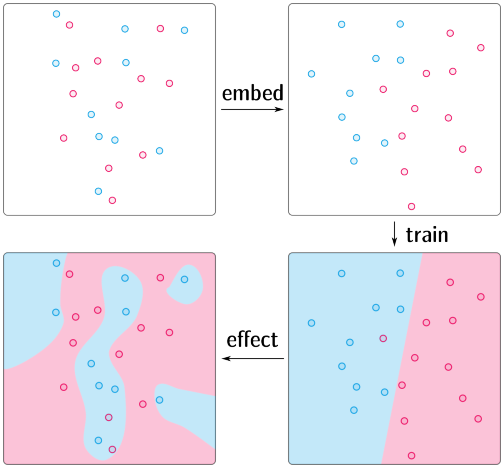


Illustration of a non-linear model. It works well with non-linear data like text, images, etc.

Illustration of Embedding



Top left: original data, not lineary separable. **Top right**: embedded data, now more easily seprable. **Bottom right**: linear model learned in embedding space. **Bottom left**: decision surface of the same model in input (linear) space.

- A **self-explainable neural network** has the form:

$$p_\theta(1 \mid \mathbf{x}) = \sigma\Big(\underbrace{\sum_i w_i(\mathbf{x})\phi_i(\mathbf{x})}_{\text{“score” of } \mathbf{x}}\Big)$$

where:

- $\phi : \mathbb{R}^d \to \mathbb{R}^k$ embeds inputs into feature space
- $\mathbf{w} : \mathbb{R}^d \to \mathbb{R}^k$ computes a weight vector for each input
- $\mathbf{w}(\mathbf{x})$ is regularized to **vary slowly** w.r.t. $\mathbf{x}$

- Defines **a different linear model for every** $\mathbf{x} \in \mathbb{R}^d$

- Linear models associated to nearby inputs $\mathbf{x}$ **encouraged to be similar**, i.e., in the neighborhood of any $\mathbf{x}_0$ there exists a constant vector $\mathbf{w}_0$ that depends only on $\mathbf{x}_0$ and a “large enough” $\alpha > 0$ such that:

$$\sum_i w_i(\mathbf{x}')\phi_i(\mathbf{x}') \approx \sum_i w_{0i}\phi_i(\mathbf{x}_0) \qquad \text{for all } \mathbf{x}' \text{ that are closer than } \alpha \text{ to } \mathbf{x}_0$$

- If $\mathbf{w}(\mathbf{x}) \equiv \mathbf{w}$ is **constant** w.r.t. $\mathbf{x}$, we obtain a linear model again

- A **self-explainable neural network** has the form:

$$p_\theta(1 \mid \mathbf{x}) = \sigma\Big(\underbrace{\sum_i w_i(\mathbf{x})\phi_i(\mathbf{x})}_{\text{“score” of } \mathbf{x}}\Big)$$

where:

- $\phi : \mathbb{R}^d \to \mathbb{R}^k$ embeds inputs into feature space
- $\mathbf{w} : \mathbb{R}^d \to \mathbb{R}^k$ computes a weight vector for each input
- $\mathbf{w}(\mathbf{x})$ is regularized to **vary slowly** w.r.t. $\mathbf{x}$

- Defines **a different linear model for every** $\mathbf{x} \in \mathbb{R}^d$

- Linear models associated to nearby inputs $\mathbf{x}$ **encouraged to be similar**, i.e., in the neighborhood of any $\mathbf{x}_0$ there exists a constant vector $\mathbf{w}_0$ that depends only on $\mathbf{x}_0$ and a “large enough” $\alpha > 0$ such that:

$$\sum_i w_i(\mathbf{x}')\phi_i(\mathbf{x}') \approx \sum_i w_{0i}\phi_i(\mathbf{x}_0) \qquad \text{for all } \mathbf{x}' \text{ that are closer than } \alpha \text{ to } \mathbf{x}_0$$

- If $\mathbf{w}(\mathbf{x}) \equiv \mathbf{w}$ is **constant** w.r.t. $\mathbf{x}$, we obtain a linear model again

- A **self-explainable neural network** has the form:

$$p_\theta(1 \mid \mathbf{x}) = \sigma\Big(\underbrace{\sum_i w_i(\mathbf{x})\phi_i(\mathbf{x})}_{\text{“score” of } \mathbf{x}}\Big)$$

where:

- $\phi : \mathbb{R}^d \to \mathbb{R}^k$ embeds inputs into feature space
- $\mathbf{w} : \mathbb{R}^d \to \mathbb{R}^k$ computes a weight vector for each input
- $\mathbf{w}(\mathbf{x})$ is regularized to **vary slowly** w.r.t. $\mathbf{x}$

- Defines **a different linear model for every $\mathbf{x} \in \mathbb{R}^d$**

- Linear models associated to nearby inputs $\mathbf{x}$ **encouraged to be similar**, i.e., in the neighborhood of any $\mathbf{x}_0$ there exists a constant vector $\mathbf{w}_0$ that depends only on $\mathbf{x}_0$ and a “large enough” $\alpha > 0$ such that:

$$\sum_i w_i(\mathbf{x}')\phi_i(\mathbf{x}') \approx \sum_i w_{0i}\phi_i(\mathbf{x}_0) \qquad \text{for all } \mathbf{x}' \text{ that are closer than } \alpha \text{ to } \mathbf{x}_0$$

- If $\mathbf{w}(\mathbf{x}) \equiv \mathbf{w}$ is **constant** w.r.t. $\mathbf{x}$, we obtain a linear model again

- A **self-explainable neural network** has the form:

$$p_\theta(1 \mid \mathbf{x}) = \sigma\Big(\underbrace{\sum_i w_i(\mathbf{x})\phi_i(\mathbf{x})}_{\text{“score” of } \mathbf{x}}\Big)$$

where:

- $\phi : \mathbb{R}^d \to \mathbb{R}^k$ embeds inputs into feature space
- $\mathbf{w} : \mathbb{R}^d \to \mathbb{R}^k$ computes a weight vector for each input
- $\mathbf{w}(\mathbf{x})$ is regularized to **vary slowly** w.r.t. $\mathbf{x}$

- Defines **a different linear model for every $\mathbf{x} \in \mathbb{R}^d$**

- Linear models associated to nearby inputs $\mathbf{x}$ **encouraged to be similar**, i.e., in the neighborhood of any $\mathbf{x}_0$ there exists a constant vector $\mathbf{w}_0$ that depends only on $\mathbf{x}_0$ and a “large enough” $\alpha > 0$ such that:

$$\sum_i w_i(\mathbf{x}')\phi_i(\mathbf{x}') \approx \sum_i w_{0i}\phi_i(\mathbf{x}_0) \qquad \text{for all } \mathbf{x}' \text{ that are closer than } \alpha \text{ to } \mathbf{x}_0$$

- If $\mathbf{w}(\mathbf{x}) \equiv \mathbf{w}$ is **constant** w.r.t. $\mathbf{x}$, we obtain a linear model again

- A **self-explainable neural network** has the form:

$$p_\theta(1 \mid \mathbf{x}) = \sigma\Big(\underbrace{\sum_i w_i(\mathbf{x})\phi_i(\mathbf{x})}_{\text{“score” of } \mathbf{x}}\Big)$$

where:

- $\phi : \mathbb{R}^d \to \mathbb{R}^k$ embeds inputs into feature space
- $\mathbf{w} : \mathbb{R}^d \to \mathbb{R}^k$ computes a weight vector for each input
- $\mathbf{w}(\mathbf{x})$ is regularized to **vary slowly** w.r.t. $\mathbf{x}$

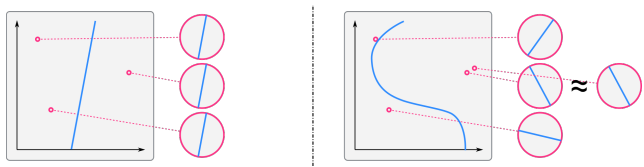
- Defines **a different linear model for every $\mathbf{x} \in \mathbb{R}^d$**

- Linear models associated to nearby inputs $\mathbf{x}$ **encouraged to be similar**, i.e., in the neighborhood of any $\mathbf{x}_0$ there exists a constant vector $\mathbf{w}_0$ that depends only on $\mathbf{x}_0$ and a “large enough” $\alpha > 0$ such that:

$$\sum_i w_i(\mathbf{x}')\phi_i(\mathbf{x}') \approx \sum_i w_{0i}\phi_i(\mathbf{x}_0) \qquad \text{for all } \mathbf{x}' \text{ that are closer than } \alpha \text{ to } \mathbf{x}_0$$

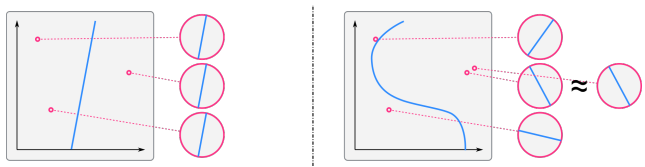
- If $\mathbf{w}(\mathbf{x}) \equiv \mathbf{w}$ is **constant** w.r.t. $\mathbf{x}$, we obtain a linear model again

- **Left**: a linear model. Notice that the weights $\mathbf{w}$ are constant everywhere.
- **Right**: a SENN. Notice that **locally** the weights $\mathbf{w}(\mathbf{x})$ are almost identical!



- SENNs are stable locally (**interpretability**) but flexible globally (**large capacity**)

- **Left**: a linear model. Notice that the weights $\mathbf{w}$ are constant everywhere.
- **Right**: a SENN. Notice that **locally** the weights $\mathbf{w}(\mathbf{x})$ are almost identical!



- SENNs are stable locally (**interpretability**) but flexible globally (**large capacity**)

Learning $\mathbf{w}(\mathbf{x})$

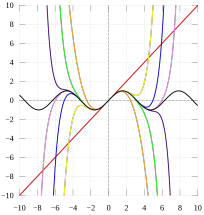
- How to ensure that $\mathbf{w}(\mathbf{x})$ is "locally linear"?

Taylor's approximation for vector-valued functions

Let $\mathbf{w}(\mathbf{x})$ be a vector-valued function of a vector input $\mathbf{x}$. Taylor's theorem implies that $\mathbf{w}$ can be approximated around any $\mathbf{x}_0$ as:

$$\mathbf{w}(\mathbf{x}) = \mathbf{w}(\mathbf{x}_0) + \underbrace{J(\mathbf{x} - \mathbf{x}_0)}_{\text{first-order term}} + \underbrace{\ldots}_{\text{quadratic+ terms}}$$

where J is the **matrix** of derivatives $J_{ab} = \frac{\partial w_a}{\partial x_b}$.



Taylor decomposition of a one-dimensional function, namely $\sin x$. The original function can be viewed as a (weighted) sum of the 1st, 2nd, 3rd, etc. **derivatives** of the function.

Credits: Wikimedia.

- The approximation is actually **exact for linear functions**:

$$\mathbf{w}^\top \mathbf{x} = \mathbf{w}^\top \mathbf{x}_0 + J(\mathbf{x} - \mathbf{x}_0)$$

- If we want $\mathbf{w}(\mathbf{x})$ to behave like a linear function we should minimize the contribution of the **quadratic term**, but doing so directly is challenging.

Idea: *regularize the model to approximate its own first-order Taylor expansion*

A SENN with **parameters** θ (including the params of $\mathbf{w}(\mathbf{x})$ and of $\phi(\mathbf{x})$) is trained by minimizing:

$$\frac{1}{|S|} \sum_{(\mathbf{x},y)\in S} - \log p_\theta(y \mid \mathbf{x}) + \lambda \cdot \Omega(\theta, \mathbf{x})$$

where the **regularizer** Ω penalizes $\mathbf{w}(\mathbf{x})$ for **deviations from linearity**:

$$\Omega(\theta, \mathbf{x}) := \| \underbrace{\nabla_\mathbf{x} p_\theta(1 \mid \mathbf{x})}_{\text{neural net analogue of weights}} - \underbrace{J_\mathbf{x}^\phi \, \theta(\mathbf{x})}_{\text{if } f \text{ were linear}} \|$$

and $\lambda > 0$ trades off between performance and non-linearity.

- Conceptually similar to tree-regularization, but with **linear models** in place of DTs. It is actually much faster because the regularizer does **not** require to learn DTs during training & Jacobian can be computed relatively quickly using autodiff packages.

Idea: *regularize the model to approximate its own first-order Taylor expansion*

A SENN with **parameters** θ (including the params of $\mathbf{w}(\mathbf{x})$ and of $\phi(\mathbf{x})$) is trained by minimizing:

$$\frac{1}{|S|} \sum_{(\mathbf{x},y)\in S} - \log p_\theta(y \mid \mathbf{x}) + \lambda \cdot \Omega(\theta, \mathbf{x})$$

where the **regularizer** Ω penalizes $\mathbf{w}(\mathbf{x})$ for **deviations from linearity**:

$$\Omega(\theta, \mathbf{x}) := \| \underbrace{\nabla_\mathbf{x} p_\theta(1 \mid \mathbf{x})}_{\text{neural net analogue of weights}} - \underbrace{J_\mathbf{x}^\phi \boldsymbol{\theta}(\mathbf{x})}_{\text{if } f \text{ were linear}} \|$$

and $\lambda > 0$ trades off between performance and non-linearity.

- Conceptually similar to tree-regularization, but with **linear models** in place of DTs. It is actually much faster because the regularizer does **not** require to learn DTs during training & Jacobian can be computed relatively quickly using autodiff packages.

Idea: *regularize the model to approximate its own first-order Taylor expansion*

A SENN with **parameters** θ (including the params of $\mathbf{w}(\mathbf{x})$ and of $\phi(\mathbf{x})$) is trained by minimizing:

$$\frac{1}{|S|} \sum_{(\mathbf{x},y)\in S} - \log p_\theta(y \mid \mathbf{x}) + \lambda \cdot \Omega(\theta, \mathbf{x})$$

where the **regularizer** Ω penalizes $\mathbf{w}(\mathbf{x})$ for **deviations from linearity**:

$$\Omega(\theta, \mathbf{x}) := \| \underbrace{\nabla_\mathbf{x} p_\theta(1 \mid \mathbf{x})}_{\text{neural net analogue of weights}} - \underbrace{J_\mathbf{x}^\phi \boldsymbol{\theta}(\mathbf{x})}_{\text{if } f \text{ were linear}} \|$$

and $\lambda > 0$ trades off between performance and non-linearity.

- Conceptually similar to tree-regularization, but with **linear models** in place of DTs. It is actually much faster because the regularizer does **not** require to learn DTs during training & Jacobian can be computed relatively quickly using autodiff packages.

Learning the Embedding Function $\phi(\mathbf{x})$

Idea: *learn to map* $\mathbf{x}$ *to* **interpretable** *concepts* ϕ. Strict requirement! Recall that an explanation looks like:

$$\big(w_1(\mathbf{x}) : \phi_1(\mathbf{x}), \quad \ldots, w_d(\mathbf{x}) : \phi_n(\mathbf{x})\big)$$

If $\phi_i(\mathbf{x})$ has clear semantics (e.g., "document $\mathbf{x}$ is about `politics`") this is a valid explanation, otherwise (e.g., for BERT embeddings) it is not!

A minimal set of **desiderata**:

1. **Fidelity**: the representation of $\mathbf{x}$ in terms of concepts should **preserve relevant information**
2. **Diversity**: inputs should be representable with **few**, **non-overlapping** concepts
3. **Grounding**: concepts should have an immediate **human-understandable** interpretation.

This is a **very rough and incomplete** list.

Remark: **nobody** knows how to formalize/implement the last desideratum properly!

Learning the Embedding Function $\phi(\mathbf{x})$

Idea: *learn to map* $\mathbf{x}$ *to* **interpretable** *concepts* ϕ. Strict requirement! Recall that an explanation looks like:

$$\big(w_1(\mathbf{x}) : \phi_1(\mathbf{x}), \quad \ldots, w_d(\mathbf{x}) : \phi_n(\mathbf{x})\big)$$

If $\phi_i(\mathbf{x})$ has clear semantics (e.g., "document $\mathbf{x}$ is about `politics`") this is a valid explanation, otherwise (e.g., for BERT embeddings) it is not!

A minimal set of **desiderata**:

1. **Fidelity**: the representation of $\mathbf{x}$ in terms of concepts should **preserve relevant information**
2. **Diversity**: inputs should be representable with **few**, **non-overlapping** concepts
3. **Grounding**: concepts should have an immediate **human-understandable** interpretation.

This is a **very rough and incomplete** list.

Remark: **nobody** knows how to formalize/implement the last desideratum properly!

Learning the Embedding Function $\phi(\mathbf{x})$

Idea: *learn to map* $\mathbf{x}$ *to* **interpretable** *concepts* ϕ. Strict requirement! Recall that an explanation looks like:

$$\big(w_1(\mathbf{x}) : \phi_1(\mathbf{x}), \quad \ldots, w_d(\mathbf{x}) : \phi_n(\mathbf{x})\big)$$

If $\phi_i(\mathbf{x})$ has clear semantics (e.g., "document $\mathbf{x}$ is about `politics`") this is a valid explanation, otherwise (e.g., for BERT embeddings) it is not!

A minimal set of **desiderata**:

1. **Fidelity**: the representation of $\mathbf{x}$ in terms of concepts should **preserve relevant information**
2. **Diversity**: inputs should be representable with **few**, **non-overlapping** concepts
3. **Grounding**: concepts should have an immediate **human-understandable** interpretation.

This is a **very rough and incomplete** list.

Remark: **nobody** knows how to formalize/implement the last desideratum properly!

- There are a few alternatives. One is to assume that $\phi(\cdot)$ is defined **manually by a domain expert**:

Example

Consider a medical diagnosis setting. A medical doctor could tell you that `lorazepam` is an important feature for predicting `clinical depression`. This can be modelled as a feature of the form:

$$\phi_3(\mathbf{x}) = \mathbb{1}(\text{the clinical record } \mathbf{x} \text{ reports administration of } \texttt{lorazepam})$$

This process makes perfect sense for **tabular data**.

- An alternative useful for non-tabular data is to learn $\phi(\cdot)$ **automatically from the data beforehand**:

Example

Train a convolutional neural network to classify ImageNet (1000 classes including many common objects) and then use the predictions made by the model to define 1000 different features, one for each class.

- Or, ideally, learn $\phi(\cdot)$ **jointly with the rest of the model**. How?

- There are a few alternatives. One is to assume that $\phi(\cdot)$ is defined **manually by a domain expert**:

Example

Consider a medical diagnosis setting. A medical doctor could tell you that `lorazepam` is an important feature for predicting `clinical depression`. This can be modelled as a feature of the form:

$$\phi_3(\mathbf{x}) = \mathbb{1}(\text{the clinical record } \mathbf{x} \text{ reports administration of } \texttt{lorazepam})$$

This process makes perfect sense for **tabular data**.

- An alternative useful for non-tabular data is to learn $\phi(\cdot)$ **automatically from the data beforehand**:

Example

Train a convolutional neural network to classify ImageNet (1000 classes including many common objects) and then use the predictions made by the model to define 1000 different features, one for each class.

- Or, ideally, learn $\phi(\cdot)$ **jointly with the rest of the model**. How?

- There are a few alternatives. One is to assume that $\phi(\cdot)$ is defined **manually by a domain expert**:

Example

Consider a medical diagnosis setting. A medical doctor could tell you that `lorazepam` is an important feature for predicting `clinical depression`. This can be modelled as a feature of the form:

$$\phi_3(\mathbf{x}) = \mathbb{1}(\text{the clinical record } \mathbf{x} \text{ reports administration of } \texttt{lorazepam})$$

This process makes perfect sense for **tabular data**.

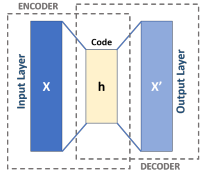
- An alternative useful for non-tabular data is to learn $\phi(\cdot)$ **automatically from the data beforehand**:

Example

Train a convolutional neural network to classify ImageNet (1000 classes including many common objects) and then use the predictions made by the model to define 1000 different features, one for each class.

- Or, ideally, learn $\phi(\cdot)$ **jointly with the rest of the model**. How?

- Model $\phi(\cdot)$ using an **autoencoder**:



An **autoencoder** is defined as an encoder-decoder pair (ϕ, ψ):

$$\phi : \mathbb{R}^d \to \mathbb{R}^k \qquad \psi : \mathbb{R}^k \to \mathbb{R}^d$$

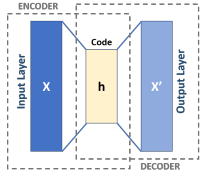
Encoder and decoder are trained jointly to minimize *reconstruction loss* $\ell_{\text{rec}}(\mathbf{x}, \mathbf{x}') = \sum_{j \in [d]} (x_i - x_i')^2$

The idea is to learn the autoencoder **end-to-end with the SENN** by minimizing:

$$\frac{1}{|S|} \sum_{(\mathbf{x},y) \in S} \left\{ -p_\theta(y \mid \mathbf{x}) + \lambda \cdot \Omega(\theta, \mathbf{x}) + \lambda' \cdot \ell_{\text{rec}}(\mathbf{x}, \psi(\phi(\mathbf{x}))) \right\}$$

- This encourages ϕ to satisfy **fidelity**, i.e., preserving both task-relevant information (because of the cross-entropy loss) and instance-relevant information (because of ℓ_{rec})

- Model $\phi(\cdot)$ using an **autoencoder**:



An **autoencoder** is defined as an encoder-decoder pair (ϕ, ψ):

$$\phi : \mathbb{R}^d \to \mathbb{R}^k \qquad \psi : \mathbb{R}^k \to \mathbb{R}^d$$

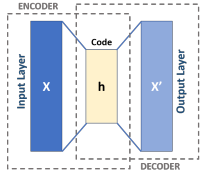
Encoder and decoder are trained jointly to minimize *reconstruction loss* $\ell_{\text{rec}}(\mathbf{x}, \mathbf{x}') = \sum_{j \in [d]} (x_i - x_i')^2$

The idea is to learn the autoencoder **end-to-end with the SENN** by minimizing:

$$\frac{1}{|S|} \sum_{(\mathbf{x},y) \in S} \left\{ -p_\theta(y \mid \mathbf{x}) + \lambda \cdot \Omega(\theta, \mathbf{x}) + \lambda' \cdot \ell_{\text{rec}}(\mathbf{x}, \psi(\phi(\mathbf{x}))) \right\}$$

- This encourages ϕ to satisfy **fidelity**, i.e., preserving both task-relevant information (because of the cross-entropy loss) and instance-relevant information (because of ℓ_{rec})

- Model $\phi(\cdot)$ using an **autoencoder**:



An **autoencoder** is defined as an encoder-decoder pair (ϕ, ψ):

$$\phi : \mathbb{R}^d \to \mathbb{R}^k \qquad \psi : \mathbb{R}^k \to \mathbb{R}^d$$

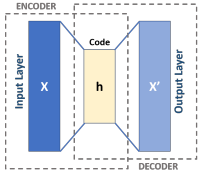
Encoder and decoder are trained jointly to minimize *reconstruction loss* $\ell_{\text{rec}}(\mathbf{x}, \mathbf{x}') = \sum_{j \in [d]} (x_i - x_i')^2$

The idea is to learn the autoencoder **end-to-end with the SENN** by minimizing:

$$\frac{1}{|S|} \sum_{(\mathbf{x},y) \in S} \left\{ -p_\theta(y \mid \mathbf{x}) + \lambda \cdot \Omega(\theta, \mathbf{x}) + \lambda' \cdot \ell_{\text{rec}}(\mathbf{x}, \psi(\phi(\mathbf{x}))) \right\}$$

- This encourages ϕ to satisfy **fidelity**, i.e., preserving both task-relevant information (because of the cross-entropy loss) and instance-relevant information (because of ℓ_{rec})

- Model $\phi(\cdot)$ using an **autoencoder**:



An **autoencoder** is defined as an encoder-decoder pair (ϕ, ψ):

$$\phi : \mathbb{R}^d \to \mathbb{R}^k \qquad \psi : \mathbb{R}^k \to \mathbb{R}^d$$

Encoder and decoder are trained jointly to minimize *reconstruction loss* $\ell_{\text{rec}}(\mathbf{x}, \mathbf{x}') = \sum_{j \in [d]} (x_i - x_i')^2$

The idea is to learn the autoencoder **end-to-end with the SENN** by minimizing:

$$\frac{1}{|S|} \sum_{(\mathbf{x},y) \in S} \left\{ -p_\theta(y \mid \mathbf{x}) + \lambda \cdot \Omega(\theta, \mathbf{x}) + \lambda' \cdot \ell_{\text{rec}}(\mathbf{x}, \psi(\phi(\mathbf{x}))) \right\}$$

- This encourages ϕ to satisfy **fidelity**, i.e., preserving both task-relevant information (because of the cross-entropy loss) and instance-relevant information (because of ℓ_{rec})

The complete architecture of a SENN is:

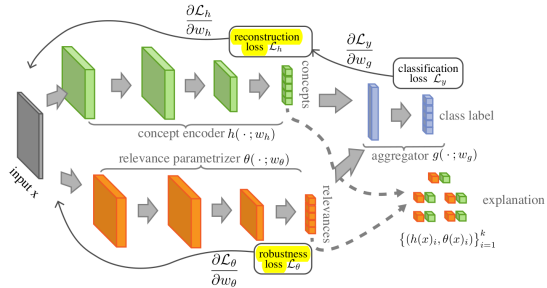


Figure 1: A SENN consists of three components: a **concept encoder** (green) that transforms the input into a small set of interpretable basis features; an **input-dependent parametrizer** (orange) that generates relevance scores; and an **aggregation function** that combines to produce a prediction. The robustness loss on the parametrizer encourages the full model to behave locally as a linear function on $h(x)$ with parameters $\theta(x)$, yielding immediate interpretation of both concepts and relevances.

Recall that during training we minimize:

$$\frac{1}{|S|}\sum_{(\mathbf{x},y)\in S}\left\{-p_\theta(y \mid \mathbf{x}) + \lambda\cdot\Omega(\theta) + \lambda'\cdot\ell_{\text{rec}}(\mathbf{x},\psi(\phi(\mathbf{x})))\right\}, \qquad \ell_{\text{rec}}(\mathbf{x},\mathbf{x}') = \sum_{j\in[d]}(x_i - x_i')^2$$

- Extra elements:
  - **Diversity**: encourage sparse concept activations by adding $\lambda''\cdot\|\phi(\mathbf{x})\|_1$ to the loss
  - **Grounding**: represent learned concepts $\phi_j(\mathbf{x})$ using **concrete examples**.
    * A set of **concrete prototypes**, i.e., training examples that maximally activate them:

$$P^{(j)} = \underset{P\subseteq S:|P|=p}{\text{argmax}} \sum_{\mathbf{x}\in P}\phi_j(\mathbf{x})$$

    * **synthetic prototypes**, i.e., inputs $\mathbf{x}$ that maximally activate one concept without activating the others:

$$\mathbf{x}^{(j)} = \underset{\mathbf{x}\in\mathbb{R}^d}{\text{argmax}}\ \phi_j(\mathbf{x}) - \sum_{k\neq j}\phi_j(\mathbf{x})$$

    In practice, approximated using gradient ascent or similar techniques.

Recall that during training we minimize:

$$\frac{1}{|S|}\sum_{(\mathbf{x},y)\in S}\left\{-p_\theta(y \mid \mathbf{x}) + \lambda\cdot\Omega(\theta) + \lambda'\cdot\ell_{\text{rec}}(\mathbf{x},\psi(\phi(\mathbf{x})))\right\}, \qquad \ell_{\text{rec}}(\mathbf{x},\mathbf{x}') = \sum_{j\in[d]}(x_i - x_i')^2$$

- Extra elements:
  - **Diversity**: encourage sparse concept activations by adding $\lambda''\cdot\|\phi(\mathbf{x})\|_1$ to the loss
  - **Grounding**: represent learned concepts $\phi_j(\mathbf{x})$ using **concrete examples**.
    * A set of **concrete prototypes**, i.e., training examples that maximally activate them:

$$P^{(j)} = \operatorname*{argmax}_{P\subseteq S:|P|=p}\sum_{\mathbf{x}\in P}\phi_j(\mathbf{x})$$

    * **synthetic prototypes**, i.e., inputs $\mathbf{x}$ that maximally activate one concept without activating the others:

$$\mathbf{x}^{(j)} = \operatorname*{argmax}_{\mathbf{x}\in\mathbb{R}^d}\ \phi_j(\mathbf{x}) - \sum_{k\neq j}\phi_j(\mathbf{x})$$

    In practice, approximated using gradient ascent or similar techniques.

Recall that during training we minimize:

$$\frac{1}{|S|}\sum_{(\mathbf{x},y)\in S}\left\{-p_\theta(y \mid \mathbf{x}) + \lambda\cdot\Omega(\theta) + \lambda'\cdot\ell_{\text{rec}}(\mathbf{x},\psi(\phi(\mathbf{x})))\right\}, \qquad \ell_{\text{rec}}(\mathbf{x},\mathbf{x}') = \sum_{j\in[d]}(x_i - x_i')^2$$

- Extra elements:
  - **Diversity**: encourage sparse concept activations by adding $\lambda''\cdot\|\phi(\mathbf{x})\|_1$ to the loss
  - **Grounding**: represent learned concepts $\phi_j(\mathbf{x})$ using **concrete examples**.

    * A set of **concrete prototypes**, i.e., training examples that maximally activate them:

$$P^{(j)} = \underset{P\subseteq S:|P|=p}{\text{argmax}} \sum_{\mathbf{x}\in P}\phi_j(\mathbf{x})$$

    * **synthetic prototypes**, i.e., inputs $\mathbf{x}$ that maximally activate one concept without activating the others:

$$\mathbf{x}^{(j)} = \underset{\mathbf{x}\in\mathbb{R}^d}{\text{argmax}}\ \phi_j(\mathbf{x}) - \sum_{k\neq j}\phi_j(\mathbf{x})$$

    In practice, approximated using gradient ascent or similar techniques.

Recall that during training we minimize:

$$\frac{1}{|S|}\sum_{(\mathbf{x},y)\in S}\left\{-p_\theta(y \mid \mathbf{x}) + \lambda\cdot\Omega(\theta) + \lambda'\cdot\ell_{\text{rec}}(\mathbf{x},\psi(\boldsymbol{\phi}(\mathbf{x})))\right\}, \qquad \ell_{\text{rec}}(\mathbf{x},\mathbf{x}') = \sum_{j\in[d]}(x_i - x_i')^2$$

- Extra elements:
  - **Diversity**: encourage sparse concept activations by adding $\lambda''\cdot\|\boldsymbol{\phi}(\mathbf{x})\|_1$ to the loss
  - **Grounding**: represent learned concepts $\phi_j(\mathbf{x})$ using **concrete examples**.

    * A set of **concrete prototypes**, i.e., training examples that maximally activate them:

$$P^{(j)} = \underset{P\subseteq S:|P|=p}{\operatorname{argmax}} \sum_{\mathbf{x}\in P}\phi_j(\mathbf{x})$$

    * **synthetic prototypes**, i.e., inputs $\mathbf{x}$ that maximally activate one concept without activating the others:

$$\mathbf{x}^{(j)} = \underset{\mathbf{x}\in\mathbb{R}^d}{\operatorname{argmax}}\ \phi_j(\mathbf{x}) - \sum_{k\neq j}\phi_j(\mathbf{x})$$

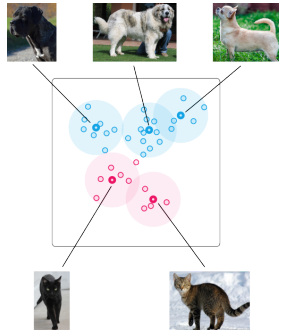
    In practice, approximated using gradient ascent or similar techniques.

Figure 2: Learned prototypes and criticisms from Imagenet dataset (two types of dog breeds)

Is there a more **direct** way of incorporating prototypes and representation learning in an interpretable manner?

Prototypes + Deep Learning

- A **prototype** is an example that is prototypical of a certain class.



- **Example**: in a `dog` vs. `cat` image classification problem, the prototypes for the dog class correspond to prototypical images of dogs (e.g., a chihuahua, a mastiffs, ...) that have "average features".

- Formally, a prototype is an example that is **close** (or **similar**) to many examples of the corresponding class, s.t. taken together they manage to "cover" all examples of that class. Distance is computed in, e.g., embedding space.

They can be found by clustering the data of a given class, for instance using k-means or other clustering algorithms.

Prototypical Networks

- What about **prototypical networks** (ProtoNets)? [Snell et al., 2017]

Idea:

- Learn an **embedding function** $\phi : \mathbb{R}^d \to \mathbb{R}^c$
- Represent each class $y \in \{1, \ldots, v\}$ by its **centroid** in embedding space $\mathbf{c}^y := \frac{1}{S^y} \sum_{(\mathbf{x},k) \in S^y} \phi(\mathbf{x})$
- Fix a distance function $d(\phi, \phi')$, compute **vector of distances** from class centroids:

$$\mathbf{d} = (d(\phi(\mathbf{x}), \mathbf{c}^1), \ldots, d(\phi(\mathbf{x}), \mathbf{c}^v))$$

  The Euclidean distance $d(\phi, \phi') = \|\phi - \phi'\|_2$ works well [Snell et al., 2017]
- Predicted **probability** of $\mathbf{x}$ belonging to class y proportional to **distance from prototype of that class**:

$$p_\theta(y \mid \mathbf{x}) := \mathrm{softmax}(-\mathbf{d})_y = \frac{\exp(-d(\phi(\mathbf{x}), c^y))}{\sum_{y'} \exp(-d(\phi(\mathbf{x}), c^{y'}))}$$

- Set of **all** parameters is $\theta = \{\phi, \mathbf{c}^1, \ldots, \mathbf{c}^v\}$.

Prototypical Networks

- What about **prototypical networks** (ProtoNets)? [Snell et al., 2017]

Idea:

- Learn an **embedding function** $\boldsymbol{\phi} : \mathbb{R}^d \to \mathbb{R}^c$
- Represent each class $y \in \{1, \ldots, v\}$ by its **centroid** in embedding space $\mathbf{c}^y := \frac{1}{S^y} \sum_{(\mathbf{x},k) \in S^y} \boldsymbol{\phi}(\mathbf{x})$
- Fix a distance function $d(\boldsymbol{\phi}, \boldsymbol{\phi}')$, compute **vector of distances** from class centroids:

$$\mathbf{d} = (d(\boldsymbol{\phi}(\mathbf{x}), \mathbf{c}^1), \ldots, d(\boldsymbol{\phi}(\mathbf{x}), \mathbf{c}^v))$$

  The Euclidean distance $d(\boldsymbol{\phi}, \boldsymbol{\phi}') = \|\boldsymbol{\phi} - \boldsymbol{\phi}'\|_2$ works well [Snell et al., 2017]
- Predicted **probability** of $\mathbf{x}$ belonging to class y proportional to **distance from prototype of that class**:

$$p_\theta(y \mid \mathbf{x}) := \mathrm{softmax}(-\mathbf{d})_y = \frac{\exp(-d(\boldsymbol{\phi}(\mathbf{x}), c^y))}{\sum_{y'} \exp(-d(\boldsymbol{\phi}(\mathbf{x}), c^{y'}))}$$

- Set of **all** parameters is $\theta = \{\boldsymbol{\phi}, \mathbf{c}^1, \ldots, \mathbf{c}^v\}$.

Prototypical Networks

- What about **prototypical networks** (ProtoNets)? [Snell et al., 2017]

Idea:

- Learn an **embedding function** $\phi : \mathbb{R}^d \to \mathbb{R}^c$
- Represent each class $y \in \{1, \ldots, v\}$ by its **centroid** in embedding space $\mathbf{c}^y := \frac{1}{S^y} \sum_{(\mathbf{x},k) \in S^y} \phi(\mathbf{x})$
- Fix a distance function $d(\phi, \phi')$, compute **vector of distances** from class centroids:

$$\mathbf{d} = (d(\phi(\mathbf{x}), \mathbf{c}^1), \ldots, d(\phi(\mathbf{x}), \mathbf{c}^v))$$

  The Euclidean distance $d(\phi, \phi') = \|\phi - \phi'\|_2$ works well [Snell et al., 2017]
- Predicted **probability** of $\mathbf{x}$ belonging to class y proportional to **distance from prototype of that class**:

$$p_\theta(y \mid \mathbf{x}) := \text{softmax}(-\mathbf{d})_y = \frac{\exp(-d(\phi(\mathbf{x}), c^y))}{\sum_{y'} \exp(-d(\phi(\mathbf{x}), c^{y'}))}$$

- Set of **all** parameters is $\theta = \{\phi, \mathbf{c}^1, \ldots, \mathbf{c}^v\}$.

Prototypical Networks

- What about **prototypical networks** (ProtoNets)? [Snell et al., 2017]

Idea:

- Learn an **embedding function** $\phi : \mathbb{R}^d \to \mathbb{R}^c$
- Represent each class $y \in \{1, \ldots, v\}$ by its **centroid** in embedding space $\mathbf{c}^y := \frac{1}{S^y} \sum_{(\mathbf{x},k) \in S^y} \phi(\mathbf{x})$
- Fix a distance function $d(\phi, \phi')$, compute **vector of distances** from class centroids:

$$\mathbf{d} = (d(\phi(\mathbf{x}), \mathbf{c}^1), \ldots, d(\phi(\mathbf{x}), \mathbf{c}^v))$$

  The Euclidean distance $d(\phi, \phi') = \|\phi - \phi'\|_2$ works well [Snell et al., 2017]

- Predicted **probability** of $\mathbf{x}$ belonging to class y proportional to **distance from prototype of that class**:

$$p_\theta(y \mid \mathbf{x}) := \text{softmax}(-\mathbf{d})_y = \frac{\exp(-d(\phi(\mathbf{x}), c^y))}{\sum_{y'} \exp(-d(\phi(\mathbf{x}), c^{y'}))}$$

- Set of **all** parameters is $\theta = \{\phi, \mathbf{c}^1, \ldots, \mathbf{c}^v\}$.

Prototypical Networks

- What about **prototypical networks** (ProtoNets)? [Snell et al., 2017]

Idea:

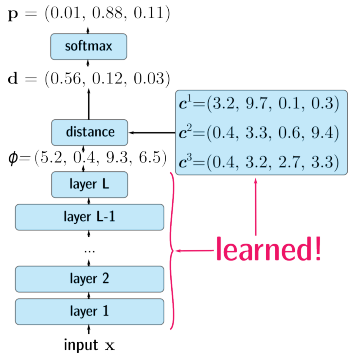
- Learn an **embedding function** $\phi : \mathbb{R}^d \to \mathbb{R}^c$
- Represent each class $y \in \{1, \ldots, v\}$ by its **centroid** in embedding space $\mathbf{c}^y := \frac{1}{S^y} \sum_{(\mathbf{x},k) \in S^y} \phi(\mathbf{x})$
- Fix a distance function $d(\phi, \phi')$, compute **vector of distances** from class centroids:

$$\mathbf{d} = (d(\phi(\mathbf{x}), \mathbf{c}^1), \ldots, d(\phi(\mathbf{x}), \mathbf{c}^v))$$

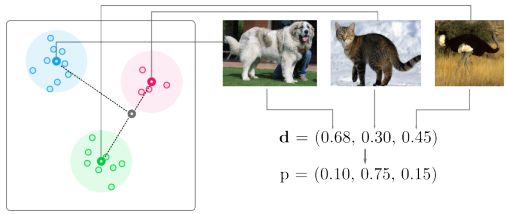
  The Euclidean distance $d(\phi, \phi') = \|\phi - \phi'\|_2$ works well [Snell et al., 2017]
- Predicted **probability** of $\mathbf{x}$ belonging to class y proportional to **distance from prototype of that class**:

$$p_\theta(y \mid \mathbf{x}) := \mathrm{softmax}(-\mathbf{d})_y = \frac{\exp(-d(\phi(\mathbf{x}), c^y))}{\sum_{y'} \exp(-d(\phi(\mathbf{x}), c^{y'}))}$$

- Set of **all** parameters is $\theta = \{\phi, \mathbf{c}^1, \ldots, \mathbf{c}^v\}$.



- *Very* simple architecture



- And also *very* explainable! The probability of each class can be traced back to the corresponding prototype!

- Fit ϕ by minimizing cross-entropy on the training set:

$$\text{argmin}_{\phi, \{\mathbf{c}^1, \ldots, \mathbf{c}^\nu\}} \; -\frac{1}{|S|} \sum_{(\mathbf{x}, y) \in S} \log p_\theta(y \mid \mathbf{x})$$

The negative log-likelihood at a training example $(\mathbf{x}, y)$ is:

$$\begin{aligned}
-\log p_\theta(y \mid \mathbf{x}) &= -\log \text{softmax}(-\mathbf{d})_y && (1)\\
&= -\log \frac{\exp(-d(\phi(\mathbf{x}), \mathbf{c}^y))}{\sum_{y'} \exp(-d(\phi(\mathbf{x}), \mathbf{c}^{y'}))} && (2)\\
&= -\{ \log \exp(-d(\phi(\mathbf{x}), \mathbf{c}^y)) - \log \textstyle\sum_{y'} \exp(-d(\phi(\mathbf{x}), \mathbf{c}^{y'})) \} && (3)\\
&= -\{ -d(\phi(\mathbf{x}), \mathbf{c}^y) - \log \textstyle\sum_{y'} \exp(-d(\phi(\mathbf{x}), \mathbf{c}^{y'})) \} && (4)\\
&= d(\phi(\mathbf{x}), \mathbf{c}^y) + \log \textstyle\sum_{y'} \exp(-d(\phi(\mathbf{x}), \mathbf{c}^{y'})) && (5)
\end{aligned}$$

The first element is the distance to the prototype of class y. The second element is the "soft maximum" of the negative distances to other classes [2]:

$$\max\{-d_1, \ldots, -d_\nu\} \leq \log \sum_{y'} \exp(-d_{y'}) \leq \max\{-d_1, \ldots, -d_\nu\} + \log(\nu)$$

Minimizing this implies (i) min. distance to true class y and (ii) approx. max. distance to other classes.

[2] See: https://en.wikipedia.org/wiki/LogSumExp

- Fit ϕ by minimizing cross-entropy on the training set:

$$\text{argmin}_{\phi, \{\mathbf{c}^1, \ldots, \mathbf{c}^\nu\}} \; -\frac{1}{|S|} \sum_{(\mathbf{x}, y) \in S} \log p_\theta(y \mid \mathbf{x})$$

The negative log-likelihood at a training example $(\mathbf{x}, y)$ is:

$$\begin{aligned}
-\log p_\theta(y \mid \mathbf{x}) &= -\log \text{softmax}(-\mathbf{d})_y && (1)\\
&= -\log \frac{\exp(-d(\phi(\mathbf{x}), \mathbf{c}^y))}{\sum_{y'} \exp(-d(\phi(\mathbf{x}), \mathbf{c}^{y'}))} && (2)\\
&= -\big\{ \log \exp(-d(\phi(\mathbf{x}), \mathbf{c}^y)) - \log \textstyle\sum_{y'} \exp(-d(\phi(\mathbf{x}), \mathbf{c}^{y'})) \big\} && (3)\\
&= -\big\{ -d(\phi(\mathbf{x}), \mathbf{c}^y) - \log \textstyle\sum_{y'} \exp(-d(\phi(\mathbf{x}), \mathbf{c}^{y'})) \big\} && (4)\\
&= d(\phi(\mathbf{x}), \mathbf{c}^y) + \log \textstyle\sum_{y'} \exp(-d(\phi(\mathbf{x}), \mathbf{c}^{y'})) && (5)
\end{aligned}$$

The first element is the distance to the prototype of class y. The second element is the "soft maximum" of the negative distances to other classes [2]:

$$\max\{-d_1, \ldots, -d_\nu\} \leq \log \sum_{y'} \exp(-d_{y'}) \leq \max\{-d_1, \ldots, -d_\nu\} + \log(\nu)$$

Minimizing this implies (i) min. distance to true class y and (ii) approx. max. distance to other classes.

[2] See: https://en.wikipedia.org/wiki/LogSumExp

- Fit ϕ by minimizing cross-entropy on the training set:

$$\text{argmin}_{\phi, \{\mathbf{c}^1, \ldots, \mathbf{c}^v\}} \; -\frac{1}{|S|} \sum_{(\mathbf{x}, y) \in S} \log p_\theta(y \mid \mathbf{x})$$

The negative log-likelihood at a training example $(\mathbf{x}, y)$ is:

$$\begin{aligned}
-\log p_\theta(y \mid \mathbf{x}) &= -\log \text{softmax}(-\mathbf{d})_y && (1)\\
&= -\log \frac{\exp(-d(\phi(\mathbf{x}), \mathbf{c}^y))}{\sum_{y'} \exp(-d(\phi(\mathbf{x}), \mathbf{c}^{y'}))} && (2)\\
&= -\big\{ \log \exp(-d(\phi(\mathbf{x}), \mathbf{c}^y)) - \log \textstyle\sum_{y'} \exp(-d(\phi(\mathbf{x}), \mathbf{c}^{y'})) \big\} && (3)\\
&= -\big\{ -d(\phi(\mathbf{x}), \mathbf{c}^y) - \log \textstyle\sum_{y'} \exp(-d(\phi(\mathbf{x}), \mathbf{c}^{y'})) \big\} && (4)\\
&= d(\phi(\mathbf{x}), \mathbf{c}^y) + \log \textstyle\sum_{y'} \exp(-d(\phi(\mathbf{x}), \mathbf{c}^{y'})) && (5)
\end{aligned}$$

The first element is the distance to the prototype of class y. The second element is the "soft maximum" of the negative distances to other classes [2]:

$$\max\{-d_1, \ldots, -d_v\} \leq \log \sum_{y'} \exp(-d_{y'}) \leq \max\{-d_1, \ldots, -d_v\} + \log(v)$$

Minimizing this implies (i) min. distance to true class y and (ii) approx. max. distance to other classes.

[2] See: https://en.wikipedia.org/wiki/LogSumExp

- Fit ϕ by minimizing cross-entropy on the training set:

$$\text{argmin}_{\phi, \{\mathbf{c}^1, \ldots, \mathbf{c}^v\}} \; -\frac{1}{|S|} \sum_{(\mathbf{x}, y) \in S} \log p_\theta(y \mid \mathbf{x})$$

The negative log-likelihood at a training example $(\mathbf{x}, y)$ is:

$$\begin{aligned}
-\log p_\theta(y \mid \mathbf{x}) &= -\log \text{softmax}(-\mathbf{d})_y && (1)\\
&= -\log \frac{\exp(-d(\phi(\mathbf{x}), \mathbf{c}^y))}{\sum_{y'} \exp(-d(\phi(\mathbf{x}), \mathbf{c}^{y'}))} && (2)\\
&= -\big\{ \log \exp(-d(\phi(\mathbf{x}), \mathbf{c}^y)) - \log \textstyle\sum_{y'} \exp(-d(\phi(\mathbf{x}), \mathbf{c}^{y'})) \big\} && (3)\\
&= -\big\{ -d(\phi(\mathbf{x}), \mathbf{c}^y) - \log \textstyle\sum_{y'} \exp(-d(\phi(\mathbf{x}), \mathbf{c}^{y'})) \big\} && (4)\\
&= d(\phi(\mathbf{x}), \mathbf{c}^y) + \log \textstyle\sum_{y'} \exp(-d(\phi(\mathbf{x}), \mathbf{c}^{y'})) && (5)
\end{aligned}$$

The first element is the distance to the prototype of class y. The second element is the "soft maximum" of the negative distances to other classes [2]:

$$\max\{-d_1, \ldots, -d_v\} \leq \log \sum_{y'} \exp(-d_{y'}) \leq \max\{-d_1, \ldots, -d_v\} + \log(v)$$

Minimizing this implies (i) min. distance to true class y and (ii) approx. max. distance to other classes.

[2] See: https://en.wikipedia.org/wiki/LogSumExp

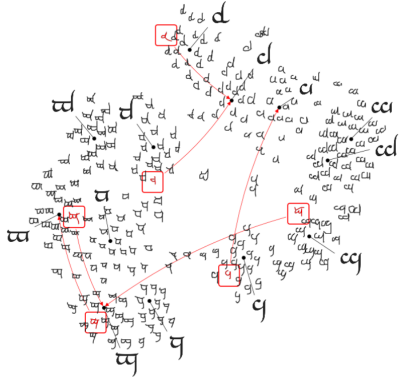
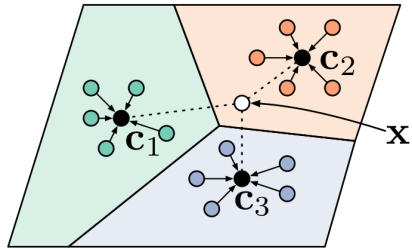


Figure 2: A t-SNE visualization of the embeddings learned by Prototypical networks on the Omniglot dataset. A subset of the Tengwar script is shown (an alphabet in the test set). Class prototypes are indicated in black. Several misclassified characters are highlighted in red along with arrows pointing to the correct prototype.

- *Very* clear results



- During training both the **space** in which the embeddings live (determined by the lower layer) *and* the prototypes $\mathbf{c}^k$ are learned jointly!

Prototypical Networks are not without **issues**:

\+ Somewhat **interpretable**:
- Each class is clearly identified by a prototype
- Each prediction can be decomposed into contributions of different prototypes

\- **Not really** interpretable:
- Class prototypes seldom correspond to concrete examples (e.g., average of several examples)
- Unclear *why* a particular prototype is relevant/similar to an example

+/- Designed for **few-shot regime**
- Only *one* prototype per class
- Works well if few examples, poorly if many

Prototypical Networks are not without **issues**:

\+ Somewhat **interpretable**:
- Each class is clearly identified by a prototype
- Each prediction can be decomposed into contributions of different prototypes

\- **Not really** interpretable:
- Class prototypes seldom correspond to concrete examples (e.g., average of several examples)
- Unclear *why* a particular prototype is relevant/similar to an example

+/- Designed for **few-shot regime**
- Only *one* prototype per class
- Works well if few examples, poorly if many

Prototypical Networks are not without **issues**:

\+ Somewhat **interpretable**:
- Each class is clearly identified by a prototype
- Each prediction can be decomposed into contributions of different prototypes

\- **Not really** interpretable:
- Class prototypes seldom correspond to concrete examples (e.g., average of several examples)
- Unclear *why* a particular prototype is relevant/similar to an example

+/- Designed for **few-shot regime**
- Only *one* prototype per class
- Works well if few examples, poorly if many

Prototype Classification Networks

Architecture of **prototype classification networks** (PCNs)

- **Autoencoder**:

  Encoder: $f : \mathbb{R}^p \to \mathbb{R}^q, \mathbf{z} := f(\mathbf{x})$ Decoder: $g : \mathbb{R}^q \to \mathbb{R}^p, \hat{\mathbf{x}} := g(\mathbf{z})$

  Learned so that $g(f(\mathbf{x})) \approx \mathbf{x}$, for instance by minimizing $\|\mathbf{x} - \hat{\mathbf{x}}\|^2$ over the training set.

- **Prototype Layer** [**new!**]
  - Memorizes m prototypes $[\mathbf{p}_1, \ldots, \mathbf{p}_m]$, with $\mathbf{p}_j \in \mathbb{R}^q$
  - Outputs squared Euclidean distances between $f(\mathbf{x})$ and each prototype:

$$p(\mathbf{z}) = (\|\mathbf{z} - \mathbf{p}_1\|^2, \ldots, \|\mathbf{z} - \mathbf{p}_m\|^2)$$

- **Dense Layer** + **Softmax**

$$p_\theta(y \mid \mathbf{x}) = \text{softmax}(Wp(f(\mathbf{x})))_y = \frac{\exp \mathbf{w}^{(y)} \cdot \mathbf{p}(f(\mathbf{x}))}{\exp \sum_{y'} \mathbf{w}^{(y')} \cdot \mathbf{p}(f(\mathbf{x}))}$$

Prototype Classification Networks

Architecture of **prototype classification networks** (PCNs)

- **Autoencoder**:

  Encoder: $f : \mathbb{R}^p \to \mathbb{R}^q, \mathbf{z} := f(\mathbf{x})$ Decoder: $g : \mathbb{R}^q \to \mathbb{R}^p, \hat{\mathbf{x}} := g(\mathbf{z})$

  Learned so that $g(f(\mathbf{x})) \approx \mathbf{x}$, for instance by minimizing $\|\mathbf{x} - \hat{\mathbf{x}}\|^2$ over the training set.

- **Prototype Layer** [**new!**]
  - Memorizes m prototypes $[\mathbf{p}_1, \ldots, \mathbf{p}_m]$, with $\mathbf{p}_j \in \mathbb{R}^q$
  - Outputs squared Euclidean distances between $f(\mathbf{x})$ and each prototype:

$$p(\mathbf{z}) = (\|\mathbf{z} - \mathbf{p}_1\|^2, \ldots, \|\mathbf{z} - \mathbf{p}_m\|^2)$$

- **Dense Layer + Softmax**

$$p_\theta(y \mid \mathbf{x}) = \mathrm{softmax}(Wp(f(\mathbf{x})))_y = \frac{\exp \mathbf{w}^{(y)} \cdot \mathbf{p}(f(\mathbf{x}))}{\exp \sum_{y'} \mathbf{w}^{(y')} \cdot \mathbf{p}(f(\mathbf{x}))}$$

Prototype Classification Networks

Architecture of **prototype classification networks** (PCNs)

- **Autoencoder**:

  Encoder: $f : \mathbb{R}^p \to \mathbb{R}^q, \mathbf{z} := f(\mathbf{x})$ Decoder: $g : \mathbb{R}^q \to \mathbb{R}^p, \hat{\mathbf{x}} := g(\mathbf{z})$

  Learned so that $g(f(\mathbf{x})) \approx \mathbf{x}$, for instance by minimizing $\|\mathbf{x} - \hat{\mathbf{x}}\|^2$ over the training set.

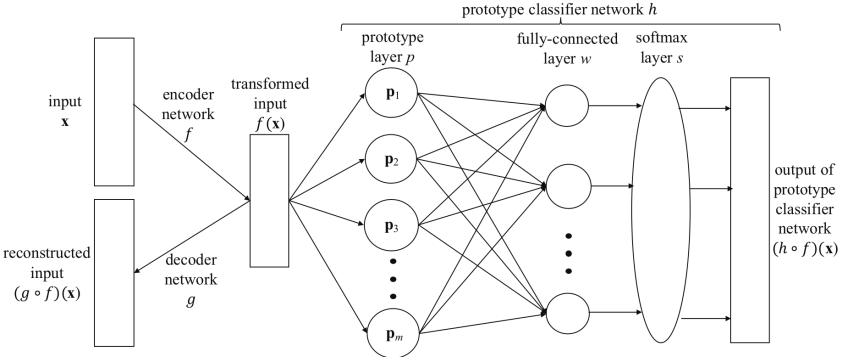
- **Prototype Layer** [**new!**]
  - Memorizes m prototypes $[\mathbf{p}_1, \ldots, \mathbf{p}_m]$, with $\mathbf{p}_j \in \mathbb{R}^q$
  - Outputs squared Euclidean distances between $f(\mathbf{x})$ and each prototype:

$$p(\mathbf{z}) = (\|\mathbf{z} - \mathbf{p}_1\|^2, \ldots, \|\mathbf{z} - \mathbf{p}_m\|^2)$$

- **Dense Layer** + **Softmax**

$$p_\theta(y \mid \mathbf{x}) = \mathrm{softmax}(Wp(f(\mathbf{x})))_y = \frac{\exp \mathbf{w}^{(y)} \cdot \mathbf{p}(f(\mathbf{x}))}{\exp \sum_{y'} \mathbf{w}^{(y')} \cdot \mathbf{p}(f(\mathbf{x}))}$$

Prototype Classification Networks



- ProtoNets:
  - Map $\mathbf{x}$ to space of emebeddings $\mathbb{R}^q$
  - Each class y is represented by **exactly one** centroid $\mathbf{c}^y \in \mathbb{R}^q$
  - Predict label based on **closest centroid**

- PCNs:
  - Map $\mathbf{x}$ to space of embeddings $\mathbb{R}^q$
  - Has a **budget of m prototypes**, not class-specific
  - Compute (squared) distance to **all prototypes**
  - Predict label based on **weighted sum of squared distances**

- PCNs are more **flexible**: possibly *multiple* prototypes per class
- PCNs recover ProtoNets if one prototype per class and W is *fixed* to $-I$

- ProtoNets:
  - Map $\mathbf{x}$ to space of emebeddings $\mathbb{R}^q$
  - Each class y is represented by **exactly one** centroid $\mathbf{c}^y \in \mathbb{R}^q$
  - Predict label based on **closest centroid**

- PCNs:
  - Map $\mathbf{x}$ to space of embeddings $\mathbb{R}^q$
  - Has a **budget of m prototypes**, not class-specific
  - Compute (squared) distance to **all prototypes**
  - Predict label based on **weighted sum of squared distances**

- PCNs are more **flexible**: possibly *multiple* prototypes per class
- PCNs recover ProtoNets if one prototype per class and W is *fixed* to $-I$

- ProtoNets:
  - Map $\mathbf{x}$ to space of emebeddings $\mathbb{R}^q$
  - Each class y is represented by **exactly one** centroid $\mathbf{c}^y \in \mathbb{R}^q$
  - Predict label based on **closest centroid**

- PCNs:
  - Map $\mathbf{x}$ to space of embeddings $\mathbb{R}^q$
  - Has a **budget of m prototypes**, not class-specific
  - Compute (squared) distance to **all prototypes**
  - Predict label based on **weighted sum of squared distances**

- PCNs are more **flexible**: possibly *multiple* prototypes per class
- PCNs recover ProtoNets if one prototype per class and W is *fixed* to $-I$

- ProtoNets:
  - Map $\mathbf{x}$ to space of emebeddings $\mathbb{R}^q$
  - Each class y is represented by **exactly one** centroid $\mathbf{c}^y \in \mathbb{R}^q$
  - Predict label based on **closest centroid**

- PCNs:
  - Map $\mathbf{x}$ to space of embeddings $\mathbb{R}^q$
  - Has a **budget of m prototypes**, not class-specific
  - Compute (squared) distance to **all prototypes**
  - Predict label based on **weighted sum of squared distances**

- PCNs are more **flexible**: possibly *multiple* prototypes per class
- PCNs recover ProtoNets if one prototype per class and W is *fixed* to $-I$

- The PCN loss is a **weighted sum** of several terms:
  - Classification loss, like the negative log-likelihood:

$$-\frac{1}{|S|}\sum_{(\mathbf{x},y)\in S}\log p_\theta(y \mid \mathbf{x}) = -\frac{1}{|S|}\sum_{(\mathbf{x},y)\in S}\sum_k \mathbb{1}(y=k)\log p_\theta(k \mid \mathbf{x})$$

  - Reconstruction loss so tha the autoencoder works as expected:

$$\frac{1}{|S|}\sum_{(\mathbf{x},y)\in S}\|\mathbf{x} - g(f(\mathbf{x}))\|^2$$

    Ensures that $\mathbf{z}$ is representative of both $\mathbf{x}$ and of y

  - **Interpretability** regularizer:

$$\frac{1}{m}\sum_{j\in[m]} \min_{(\mathbf{x},y)\in S} \|\mathbf{p}_j - f(\mathbf{x})\|^2$$

    Each prototype must be as close as possible to one training example $\rightarrow$ if decoder is smooth, decoding of prototype will be interpretable

  - **Clustering** regularizer:

$$\frac{1}{|S|}\sum_{(\mathbf{x},y)\in S} \min_{j\in[m]} \|\mathbf{p}_j - f(\mathbf{x})\|^2$$

    Each example must be as close as possible to one of the prototypes $\rightarrow$ prototypes cluster examples

- In practice min over S restricted to mini-batch.

- The PCN loss is a **weighted sum** of several terms:
  - Classification loss, like the negative log-likelihood:

$$-\frac{1}{|S|}\sum_{(\mathbf{x},y)\in S}\log p_\theta(y \mid \mathbf{x}) = -\frac{1}{|S|}\sum_{(\mathbf{x},y)\in S}\sum_k \mathbb{1}(y=k)\log p_\theta(k \mid \mathbf{x})$$

  - Reconstruction loss so tha the autoencoder works as expected:

$$\frac{1}{|S|}\sum_{(\mathbf{x},y)\in S}\|\mathbf{x} - g(f(\mathbf{x}))\|^2$$

    Ensures that $\mathbf{z}$ is representative of both $\mathbf{x}$ and of y
  - **Interpretability** regularizer:

$$\frac{1}{m}\sum_{j\in[m]} \min_{(\mathbf{x},y)\in S} \|\mathbf{p}_j - f(\mathbf{x})\|^2$$

    Each prototype must be as close as possible to one training example → if decoder is smooth, decoding of prototype will be interpretable
  - **Clustering** regularizer:

$$\frac{1}{|S|}\sum_{(\mathbf{x},y)\in S} \min_{j\in[m]} \|\mathbf{p}_j - f(\mathbf{x})\|^2$$

    Each example must be as close as possible to one of the prototypes → prototypes cluster examples

- In practice min over S restricted to mini-batch.

- The PCN loss is a **weighted sum** of several terms:
  - Classification loss, like the negative log-likelihood:

$$-\frac{1}{|S|}\sum_{(\mathbf{x},y)\in S}\log p_\theta(y \mid \mathbf{x}) = -\frac{1}{|S|}\sum_{(\mathbf{x},y)\in S}\sum_k \mathbb{1}(y=k)\log p_\theta(k \mid \mathbf{x})$$

  - Reconstruction loss so tha the autoencoder works as expected:

$$\frac{1}{|S|}\sum_{(\mathbf{x},y)\in S}\|\mathbf{x} - g(f(\mathbf{x}))\|^2$$

    Ensures that $\mathbf{z}$ is representative of both $\mathbf{x}$ and of y
  - **Interpretability** regularizer:

$$\frac{1}{m}\sum_{j\in[m]} \min_{(\mathbf{x},y)\in S} \|\mathbf{p}_j - f(\mathbf{x})\|^2$$

    Each prototype must be as close as possible to one training example $\rightarrow$ if decoder is smooth, decoding of prototype will be interpretable
  - **Clustering** regularizer:

$$\frac{1}{|S|}\sum_{(\mathbf{x},y)\in S} \min_{j\in[m]} \|\mathbf{p}_j - f(\mathbf{x})\|^2$$

    Each example must be as close as possible to one of the prototypes $\rightarrow$ prototypes cluster examples

- In practice min over S restricted to mini-batch.

- The PCN loss is a **weighted sum** of several terms:
  - Classification loss, like the negative log-likelihood:

$$-\frac{1}{|S|}\sum_{(\mathbf{x},y)\in S}\log p_\theta(y \mid \mathbf{x}) = -\frac{1}{|S|}\sum_{(\mathbf{x},y)\in S}\sum_k \mathbb{1}(y=k)\log p_\theta(k \mid \mathbf{x})$$

  - Reconstruction loss so tha the autoencoder works as expected:

$$\frac{1}{|S|}\sum_{(\mathbf{x},y)\in S}\|\mathbf{x} - g(f(\mathbf{x}))\|^2$$

    Ensures that $\mathbf{z}$ is representative of both $\mathbf{x}$ and of y
  - **Interpretability** regularizer:

$$\frac{1}{m}\sum_{j\in[m]} \min_{(\mathbf{x},y)\in S} \|\mathbf{p}_j - f(\mathbf{x})\|^2$$

    Each prototype must be as close as possible to one training example → if decoder is smooth, decoding of prototype will be interpretable
  - **Clustering** regularizer:

$$\frac{1}{|S|}\sum_{(\mathbf{x},y)\in S} \min_{j\in[m]} \|\mathbf{p}_j - f(\mathbf{x})\|^2$$

    Each example must be as close as possible to one of the prototypes → prototypes cluster examples

- In practice min over S restricted to mini-batch.

- The PCN loss is a **weighted sum** of several terms:
  - Classification loss, like the negative log-likelihood:

$$-\frac{1}{|S|}\sum_{(\mathbf{x},y)\in S}\log p_\theta(y \mid \mathbf{x}) = -\frac{1}{|S|}\sum_{(\mathbf{x},y)\in S}\sum_k \mathbb{1}(y=k)\log p_\theta(k \mid \mathbf{x})$$

  - Reconstruction loss so tha the autoencoder works as expected:

$$\frac{1}{|S|}\sum_{(\mathbf{x},y)\in S}\|\mathbf{x} - g(f(\mathbf{x}))\|^2$$

    Ensures that $\mathbf{z}$ is representative of both $\mathbf{x}$ and of y
  - **Interpretability** regularizer:

$$\frac{1}{m}\sum_{j\in[m]} \min_{(\mathbf{x},y)\in S} \|\mathbf{p}_j - f(\mathbf{x})\|^2$$

    Each prototype must be as close as possible to one training example $\rightarrow$ if decoder is smooth, decoding of prototype will be interpretable
  - **Clustering** regularizer:

$$\frac{1}{|S|}\sum_{(\mathbf{x},y)\in S} \min_{j\in[m]} \|\mathbf{p}_j - f(\mathbf{x})\|^2$$

    Each example must be as close as possible to one of the prototypes $\rightarrow$ prototypes cluster examples

- In practice min over S restricted to mini-batch.

Learned Prototypes

Figure 2: Some random images from the training set in the first row and their corresponding reconstructions in the second row.

Figure 3: 15 learned MNIST prototypes visualized in pixel space.

Learned Prototypes

Figure 9: 15 decoded prototypes for Fashion-MNIST.

Learned Prototypes

	interpretable	non-interpretable
train acc	98.2%	99.8%
test acc	93.5%	94.2%

Table 3: Car dataset accuracy.

Figure 5: Decoded prototypes when we include R_1 and R_2.

- Cars dataset – contains small B/W images of cars from different angles.
- **High performance** without entirely sacrificing interpretability.

	0	1	2	3	4	5	6	7	8	9
	-0.07	7.77	1.81	0.66	4.01	2.08	3.11	4.10	-20.45	-2.34
	2.84	3.29	1.16	1.80	-1.05	4.36	4.40	-0.71	0.97	-18.10
	-25.66	4.32	-0.23	6.16	1.60	0.94	1.82	1.56	3.98	-1.77
	-1.22	1.64	3.64	4.04	0.82	0.16	2.44	-22.36	4.04	1.78
	2.72	-0.27	-0.49	-12.00	2.25	-3.14	2.49	3.96	5.72	-1.62
	-5.52	1.42	2.36	1.48	0.16	0.43	-11.12	2.41	1.43	1.25
	4.77	2.02	2.21	-13.64	3.52	-1.32	3.01	0.18	-0.56	-1.49
	0.52	-24.16	2.15	2.63	-0.09	2.25	0.71	0.59	3.06	2.00
	0.56	-1.28	1.83	-0.53	-0.98	-0.97	-10.56	4.27	1.35	4.04
	-0.18	1.68	0.88	2.60	-0.11	-3.29	-11.20	2.76	0.52	0.75
	5.98	0.64	4.77	-1.43	3.13	-17.53	1.17	1.08	-2.27	0.78
	1.53	-5.63	-8.78	0.10	1.56	3.08	0.43	-0.36	1.69	3.49
	1.71	1.49	-13.31	-0.69	-0.38	4.55	1.72	1.59	3.18	2.19
	5.06	-0.03	0.96	4.35	-21.75	4.25	1.42	-1.27	1.64	0.78
	-1.31	-0.62	-2.69	0.96	2.36	2.83	2.76	-4.82	-4.14	4.95

Table 1: Transposed weight matrix (every entry rounded off to 2 decimal places) between the prototype layer and the softmax layer. Each row represents a prototype node whose decoded image is shown in the first column. Each column represents a digit class. The most negative weight is shaded for each prototype. In general, for each prototype, its most negative weight is towards its visual class except for the prototype in the last row.

- Interpretation of prototype-class weights for MNIST

Effect of Regularizers

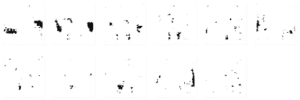


Figure 6: Decoded prototypes when we remove R_1 and R_2.

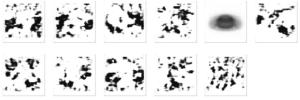


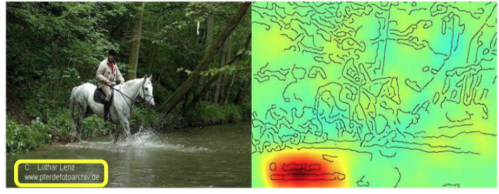
Figure 7: Decoded prototypes when we remove R_1.

- Disabling the regularizers **hinders** interpretability of the prototypes

- Is autoencoding the way to go?

- Can we go beyond concrete prototypes and look at *where* certain prototypes activate?

Horse-picture from Pascal VOC data set



Source tag present

↓

Classified as horse

No source tag present

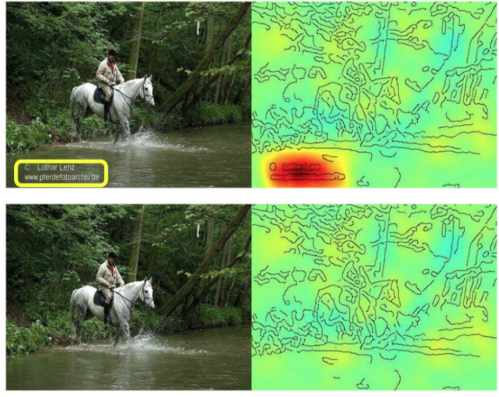
↓

Not classified as horse

- Is autoencoding the way to go?

- Can we go beyond concrete prototypes and look at *where* certain prototypes activate?

Horse-picture from Pascal VOC data set



- How would you describe why the image looks like a "clay colored sparrow"?

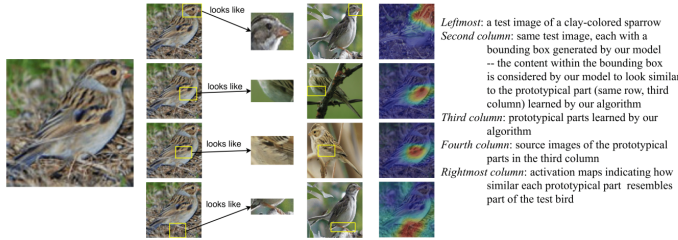


Figure 1: Image of a clay colored sparrow and how parts of it look like some learned prototypical parts of a clay colored sparrow used to classify the bird's species.

- Perhaps bird's **head** and **wing bars** *look like* those of a **prototypical** clay colored sparrow
- Radiologists compare X-ray scans with prototypical tumor images

Idea: enable models to focus on **parts** of the image and compare them with **prototypical parts** of training images from a class – reasoning of the form "this looks like that"

- How would you describe why the image looks like a "clay colored sparrow"?

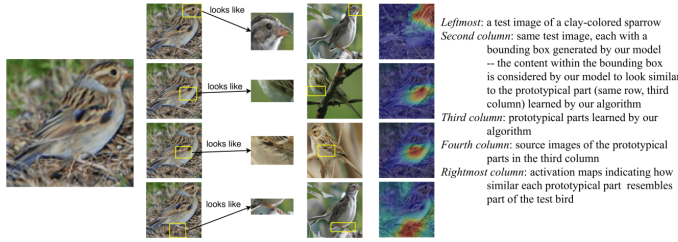


Figure 1: Image of a clay colored sparrow and how parts of it look like some learned prototypical parts of a clay colored sparrow used to classify the bird's species.

- Perhaps bird's **head** and **wing bars** *look like* those of a **prototypical** clay colored sparrow
- Radiologists compare X-ray scans with prototypical tumor images

Idea: enable models to focus on parts of the image and compare them with prototypical parts of training images from a class – reasoning of the form "this looks like that"

- How would you describe why the image looks like a "clay colored sparrow"?

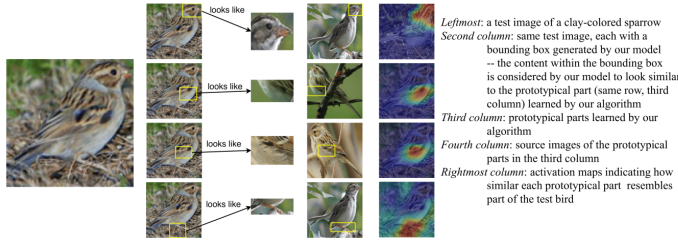
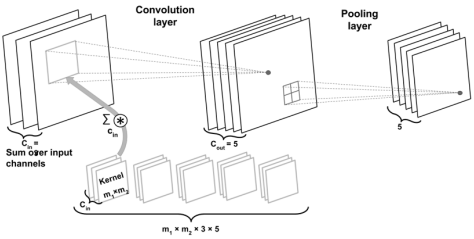


Figure 1: Image of a clay colored sparrow and how parts of it look like some learned prototypical parts of a clay colored sparrow used to classify the bird's species.

- Perhaps bird's **head** and **wing bars** *look like* those of a **prototypical** clay colored sparrow
- Radiologists compare X-ray scans with prototypical tumor images

Idea: enable models to focus on **parts** of the image and compare them with **prototypical parts** of training images from a class – reasoning of the form "this looks like that"

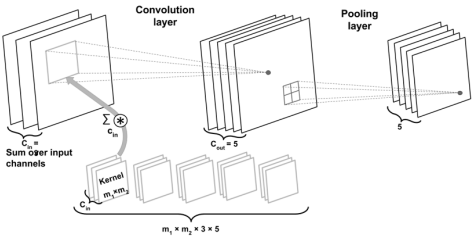
Refresher: Convolutional Filters



- **Structure**:
  - Given an **input** x of size $w \times h \times c$
  - A conv. layer has d **kernels** k_j, $j \in [d]$, each of size $w' \times h' \times c$
  - Each **kernel is convolved with the input** to obtain an output y_j of size $a \times b$, with $a = w - 2\left\lfloor \frac{w'}{2} \right\rfloor$ and $b = h - 2\left\lfloor \frac{h'}{2} \right\rfloor$
  - The outputs $y_1, \ldots, y_d$ are **stacked** to obtain the complete $a \times b \times d$ embedding y

- The size of the kernel is the **receptive field** of the convolutional layer

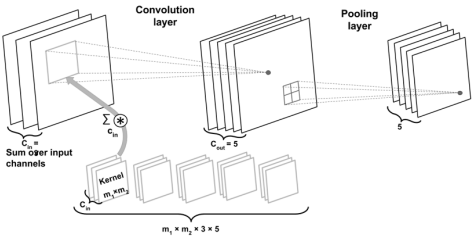
Refresher: Convolutional Filters



- **Structure**:
  - Given an **input** x of size $w \times h \times c$
  - A conv. layer has d **kernels** k_j, $j \in [d]$, each of size $w' \times h' \times c$
  - Each **kernel is convolved with the input** to obtain an output y_j of size $a \times b$, with $a = w - 2\left\lfloor \frac{w'}{2} \right\rfloor$ and $b = h - 2\left\lfloor \frac{h'}{2} \right\rfloor$
  - The outputs $y_1, \ldots, y_d$ are **stacked** to obtain the complete $a \times b \times d$ embedding y

- The size of the kernel is the **receptive field** of the convolutional layer

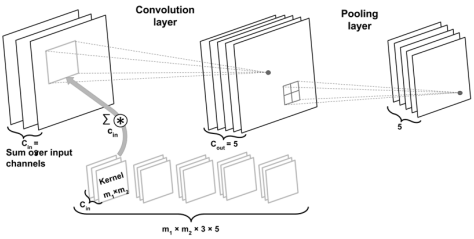
Refresher: Convolutional Filters



- **Structure**:
  - Given an **input** x of size $w \times h \times c$
  - A conv. layer has d **kernels** k_j, $j \in [d]$, each of size $w' \times h' \times c$
  - Each **kernel is convolved with the input** to obtain an output y_j of size $a \times b$, with $a = w - 2\left\lfloor \frac{w'}{2} \right\rfloor$ and $b = h - 2\left\lfloor \frac{h'}{2} \right\rfloor$
  - The outputs $y_1, \ldots, y_d$ are **stacked** to obtain the complete $a \times b \times d$ embedding y

- The size of the kernel is the receptive field of the convolutional layer

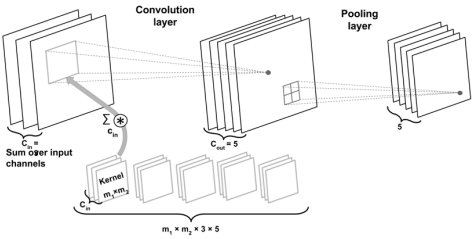
Refresher: Convolutional Filters



- **Structure**:
  - Given an **input** x of size $w \times h \times c$
  - A conv. layer has d **kernels** k_j, $j \in [d]$, each of size $w' \times h' \times c$
  - Each **kernel is convolved with the input** to obtain an output y_j of size $a \times b$, with $a = w - 2\left\lfloor \frac{w'}{2} \right\rfloor$ and $b = h - 2\left\lfloor \frac{h'}{2} \right\rfloor$
  - The outputs $y_1, \ldots, y_d$ are **stacked** to obtain the complete $a \times b \times d$ embedding y

- The size of the kernel is the **receptive field** of the convolutional layer

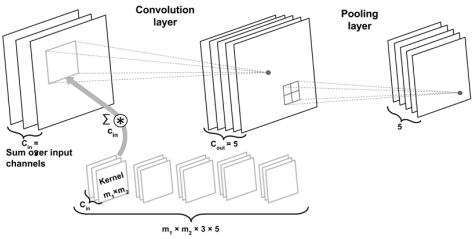
Refresher: Convolutional Networks



- Convolutional filters take an input, typically reduce its size, and output a variable number of channels (depth)
- Pooling layers behave similarly but aggregate their inputs using max or avg, and have no learnable parameters
- CNNs stack convolutional layers intermixed with pooling layers (e.g., max activations) on top of each other to produce a latent representation:

$$w \times h \times c \quad \longrightarrow \quad w' \times h' \times d$$

Refresher: Convolutional Networks



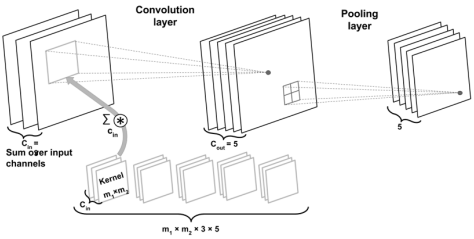
- Convolutional filters take an input, typically reduce its size, and output a variable number of channels (depth)

- Pooling layers behave similarly but aggregate their inputs using max or avg, and have no learnable parameters

- CNNs stack convolutional layers intermixed with pooling layers (e.g., max activations) on top of each other to produce a latent representation:

$$w \times h \times c \quad \longrightarrow \quad w' \times h' \times d$$

Refresher: Convolutional Networks



- Convolutional filters take an input, typically reduce its size, and output a variable number of channels (depth)

- Pooling layers behave similarly but aggregate their inputs using max or avg, and have no learnable parameters

- CNNs stack convolutional layers intermixed with pooling layers (e.g., max activations) on top of each other to produce a latent representation:

$$w \times h \times c \quad \longrightarrow \quad w' \times h' \times d$$

- Consider convolutional embeddings $\mathbf{z} = f(\mathbf{x})$:

$$w \times h \times c \quad \longrightarrow \quad w' \times h' \times d$$

with $w' \leq w$ and $h' \leq h$

- In ProtoNets and PCNs, a **prototype** $\mathbf{p} \in \mathbb{R}^{w' \times h' \times d}$ is a point in embedding space:
  - Summarizes a **set** of **examples**
  - Distance from **prototype** used as activation
  - Interpretability achieved by ensuring that $\mathbf{p}$ is "close" to **concrete example**

- In PPNets, a **part-prototype** $\mathbf{p} \in \mathbb{R}^{1 \times 1 \times d}$ is a *part* of a point in embedding space
  - Summarizes a **set** of **example parts**
  - Distance from **part-prototype** used as activation
  - Interpretability achieved by ensuring that $\mathbf{p}$ is "close" to concrete **example parts**

- Consider convolutional embeddings $\mathbf{z} = f(\mathbf{x})$:

$$w \times h \times c \quad \longrightarrow \quad w' \times h' \times d$$

with $w' \leq w$ and $h' \leq h$

- In ProtoNets and PCNs, a **prototype** $\mathbf{p} \in \mathbb{R}^{w' \times h' \times d}$ is a point in embedding space:
  - Summarizes a **set** of **examples**
  - Distance from **prototype** used as activation
  - Interpretability achieved by ensuring that $\mathbf{p}$ is "close" to **concrete example**

- In PPNets, a **part-prototype** $\mathbf{p} \in \mathbb{R}^{1 \times 1 \times d}$ is a *part* of a point in embedding space
  - Summarizes a **set** of **example parts**
  - Distance from **part-prototype** used as activation
  - Interpretability achieved by ensuring that $\mathbf{p}$ is "close" to concrete **example parts**

- Consider convolutional embeddings $\mathbf{z} = f(\mathbf{x})$:

$$w \times h \times c \quad \longrightarrow \quad w' \times h' \times d$$

with $w' \leq w$ and $h' \leq h$

- In ProtoNets and PCNs, a **prototype** $\mathbf{p} \in \mathbb{R}^{w' \times h' \times d}$ is a point in embedding space:
  - Summarizes a **set** of **examples**
  - Distance from **prototype** used as activation
  - Interpretability achieved by ensuring that $\mathbf{p}$ is "close" to **concrete example**

- In PPNets, a **part-prototype** $\mathbf{p} \in \mathbb{R}^{1 \times 1 \times d}$ is a *part* of a point in embedding space
  - Summarizes a **set** of **example parts**
  - Distance from **part-prototype** used as activation
  - Interpretability achieved by ensuring that $\mathbf{p}$ is "close" to concrete **example parts**

Part-Prototype Networks

Architecture of **part-prototype networks** (PPNets)

- **Embedding** function [**it was an autoencoder**]

$$f : \mathbb{R}^{w \times h \times c} \to \mathbb{R}^{w' \times h' \times d}$$

Loaded from a **pre-trained network**. Top layers can be **fine-tuned** while leaving the rest fixed (frozen).

- **Part-prototype Layer**
  - Memorizes m part-prototypes $[\mathbf{p}_1, \ldots, \mathbf{p}_m]$, with $\mathbf{p}_j \in \mathbb{R}^{1 \times 1 \times d}$ [**they were full prototypes**]
  - Part-prototypes are per class, $\left\lfloor \frac{m}{\nu} \right\rfloor$ for each class $y \in [\nu]$ [**they were shared**]
  - Computes **activation** of part-prototypes of each y on $\mathbf{z} = f(\mathbf{x})$:

$$\mathbf{a} = \mathbf{a}^{(1)} \circ \ldots \circ \mathbf{a}^{(\nu)} \qquad \mathbf{a}^{(y)}(\mathbf{z}) = [\mathrm{act}(\mathbf{z}, \mathbf{p}_1^{(y)}))^2, \ldots, \mathrm{act}(\mathbf{z}, \mathbf{p}_m^{(y)})^2]$$

[**it was squared L_2 distance**]

- **Dense Layer + Softmax** [**same**]

$$p_\theta(y \mid \mathbf{x}) = \mathrm{softmax}(W\mathbf{a}(f(\mathbf{x})))_y = \frac{\exp \mathbf{w}^{(y)} \cdot \mathbf{a}^{(y)}(f(\mathbf{x}))}{\exp \sum_{y'} \mathbf{w}^{(y')} \cdot \mathbf{a}^{(y')}(f(\mathbf{x}))}$$

Architecture of **part-prototype networks** (PPNets)

- **Embedding** function [**it was an autoencoder**]

$$f : \mathbb{R}^{w \times h \times c} \to \mathbb{R}^{w' \times h' \times d}$$

  Loaded from a **pre-trained network**. Top layers can be **fine-tuned** while leaving the rest fixed (frozen).

- **Part-prototype Layer**
  - Memorizes m part-prototypes $[\mathbf{p}_1, \ldots, \mathbf{p}_m]$, with $\mathbf{p}_j \in \mathbb{R}^{1 \times 1 \times d}$ [**they were full prototypes**]
  - Part-prototypes are per class, $\lfloor \frac{m}{\nu} \rfloor$ for each class $y \in [\nu]$ [**they were shared**]
  - Computes **activation** of part-prototypes of each y on $\mathbf{z} = f(\mathbf{x})$:

$$\mathbf{a} = \mathbf{a}^{(1)} \circ \ldots \circ \mathbf{a}^{(\nu)} \qquad \mathbf{a}^{(y)}(\mathbf{z}) = [\mathrm{act}(\mathbf{z}, \mathbf{p}_1^{(y)}))^2, \ldots, \mathrm{act}(\mathbf{z}, \mathbf{p}_m^{(y)})^2]$$

  [**it was squared L_2 distance**]

- **Dense Layer** + **Softmax** [**same**]

$$p_\theta(y \mid \mathbf{x}) = \mathrm{softmax}(W\mathbf{a}(f(\mathbf{x})))_y = \frac{\exp \mathbf{w}^{(y)} \cdot \mathbf{a}^{(y)}(f(\mathbf{x}))}{\exp \sum_{y'} \mathbf{w}^{(y')} \cdot \mathbf{a}^{(y')}(f(\mathbf{x}))}$$

Part-Prototype Networks

Architecture of **part-prototype networks** (PPNets)

- **Embedding** function [**it was an autoencoder**]

$$f : \mathbb{R}^{w \times h \times c} \to \mathbb{R}^{w' \times h' \times d}$$

  Loaded from a **pre-trained network**. Top layers can be **fine-tuned** while leaving the rest fixed (frozen).

- **Part-prototype Layer**
  - Memorizes m part-prototypes $[\mathbf{p}_1, \ldots, \mathbf{p}_m]$, with $\mathbf{p}_j \in \mathbb{R}^{1 \times 1 \times d}$ [**they were full prototypes**]
  - Part-prototypes are per class, $\lfloor \frac{m}{\nu} \rfloor$ for each class $y \in [\nu]$ [**they were shared**]
  - Computes **activation** of part-prototypes of each y on $\mathbf{z} = f(\mathbf{x})$:

$$\mathbf{a} = \mathbf{a}^{(1)} \circ \ldots \circ \mathbf{a}^{(\nu)} \qquad \mathbf{a}^{(y)}(\mathbf{z}) = [\mathrm{act}(\mathbf{z}, \mathbf{p}_1^{(y)}))^2, \ldots, \mathrm{act}(\mathbf{z}, \mathbf{p}_m^{(y)})^2]$$

  [**it was squared L_2 distance**]

- **Dense Layer** + **Softmax** [**same**]

$$p_\theta(y \mid \mathbf{x}) = \mathrm{softmax}(W\mathbf{a}(f(\mathbf{x})))_y = \frac{\exp \mathbf{w}^{(y)} \cdot \mathbf{a}^{(y)}(f(\mathbf{x}))}{\exp \sum_{y'} \mathbf{w}^{(y')} \cdot \mathbf{a}^{(y')}(f(\mathbf{x}))}$$

Architecture of **part-prototype networks** (PPNets)

- **Embedding** function [**it was an autoencoder**]

$$f : \mathbb{R}^{w \times h \times c} \to \mathbb{R}^{w' \times h' \times d}$$

  Loaded from a **pre-trained network**. Top layers can be **fine-tuned** while leaving the rest fixed (frozen).

- **Part-prototype Layer**
  - Memorizes m part-prototypes $[\mathbf{p}_1, \ldots, \mathbf{p}_m]$, with $\mathbf{p}_j \in \mathbb{R}^{1 \times 1 \times d}$ [**they were full prototypes**]
  - Part-prototypes are per class, $\lfloor \frac{m}{\nu} \rfloor$ for each class $y \in [\nu]$ [**they were shared**]
  - Computes **activation** of part-prototypes of each y on $\mathbf{z} = f(\mathbf{x})$:

$$\mathbf{a} = \mathbf{a}^{(1)} \circ \ldots \circ \mathbf{a}^{(\nu)} \qquad \mathbf{a}^{(y)}(\mathbf{z}) = [\mathrm{act}(\mathbf{z}, \mathbf{p}_1^{(y)}))^2, \ldots, \mathrm{act}(\mathbf{z}, \mathbf{p}_m^{(y)})^2]$$

  [**it was squared L_2 distance**]

- **Dense Layer + Softmax** [**same**]

$$p_\theta(y \mid \mathbf{x}) = \mathrm{softmax}(W\mathbf{a}(f(\mathbf{x})))_y = \frac{\exp \mathbf{w}^{(y)} \cdot \mathbf{a}^{(y)}(f(\mathbf{x}))}{\exp \sum_{y'} \mathbf{w}^{(y')} \cdot \mathbf{a}^{(y')}(f(\mathbf{x}))}$$

Part-Prototype Networks

Architecture of **part-prototype networks** (PPNets)

- **Embedding** function [**it was an autoencoder**]

$$f : \mathbb{R}^{w \times h \times c} \to \mathbb{R}^{w' \times h' \times d}$$

  Loaded from a **pre-trained network**. Top layers can be **fine-tuned** while leaving the rest fixed (frozen).

- **Part-prototype Layer**
  - Memorizes m part-prototypes $[\mathbf{p}_1, \ldots, \mathbf{p}_m]$, with $\mathbf{p}_j \in \mathbb{R}^{1 \times 1 \times d}$ [**they were full prototypes**]
  - Part-prototypes are per class, $\lfloor \frac{m}{v} \rfloor$ for each class $y \in [v]$ [**they were shared**]
  - Computes **activation** of part-prototypes of each y on $\mathbf{z} = f(\mathbf{x})$:

$$\mathbf{a} = \mathbf{a}^{(1)} \circ \ldots \circ \mathbf{a}^{(v)} \qquad \mathbf{a}^{(y)}(\mathbf{z}) = [\mathrm{act}(\mathbf{z}, \mathbf{p}_1^{(y)}))^2, \ldots, \mathrm{act}(\mathbf{z}, \mathbf{p}_m^{(y)})^2]$$

  [**it was squared L_2 distance**]

- **Dense Layer** + **Softmax** [**same**]

$$p_\theta(y \mid \mathbf{x}) = \mathrm{softmax}(W\mathbf{a}(f(\mathbf{x})))_y = \frac{\exp \mathbf{w}^{(y)} \cdot \mathbf{a}^{(y)}(f(\mathbf{x}))}{\exp \sum_{y'} \mathbf{w}^{(y')} \cdot \mathbf{a}^{(y')}(f(\mathbf{x}))}$$

- How to measure **activation** of **part**-prototype $\mathbf{p} \in \mathbb{R}^{1\times1\times d}$ on a full embedding $\mathbf{z} \in \mathbb{R}^{w'\times h'\times d}$?

  - Break down $\mathbf{z}$ into all its **pieces** $\widetilde{\mathbf{z}}$ of size $1 \times 1 \times d$, denoted:

$$\text{parts}(\mathbf{z})$$

  - Measure L_2 **distance** between $\mathbf{p}$ and each part $\widetilde{\mathbf{z}}$ of $\mathbf{z}$:

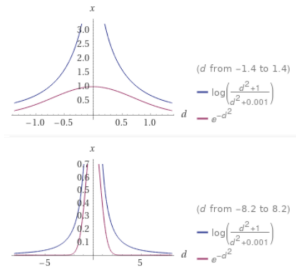
$$d(\mathbf{p}, \widetilde{\mathbf{z}}) = \|\mathbf{p} - \mathbf{z}\|$$

  - Convert distance into **activation**:

$$\text{act}(\mathbf{p}, \widetilde{\mathbf{z}}) = \log\left(\frac{d(\mathbf{p}, \widetilde{\mathbf{z}})^2 + 1}{d(\mathbf{p}, \widetilde{\mathbf{z}})^2 + \epsilon}\right)$$

  - Define activation of $\mathbf{p}$ on full embeddings $\mathbf{z}$ as maximum activation of its parts:

$$\text{act}(\mathbf{p}, \mathbf{z}) = \max_{\widetilde{\mathbf{z}} \in \text{parts}(\mathbf{z})} \text{act}(\mathbf{p}, \widetilde{\mathbf{z}})$$



Comparison between difference-of-logs and Gaussian of d:

$$\text{act}'(\mathbf{p}, \widetilde{\mathbf{z}}) = \exp\left(-\gamma \cdot d(\mathbf{p}, \widetilde{\mathbf{z}})^2\right)$$

In the plot $\epsilon = 0.001$, $\gamma = 1$

- How to measure **activation** of **part**-prototype $\mathbf{p} \in \mathbb{R}^{1\times1\times d}$ on a full embedding $\mathbf{z} \in \mathbb{R}^{w'\times h'\times d}$?

  - Break down $\mathbf{z}$ into all its **pieces** $\widetilde{\mathbf{z}}$ of size $1 \times 1 \times d$, denoted:

$$\text{parts}(\mathbf{z})$$

  - Measure L_2 **distance** between $\mathbf{p}$ and each part $\widetilde{\mathbf{z}}$ of $\mathbf{z}$:

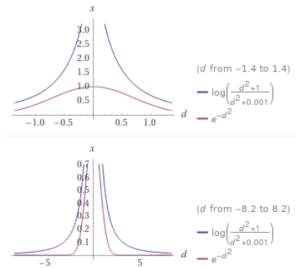
$$d(\mathbf{p}, \widetilde{\mathbf{z}}) = \|\mathbf{p} - \mathbf{z}\|$$

  - Convert distance into **activation**:

$$\text{act}(\mathbf{p}, \widetilde{\mathbf{z}}) = \log\left(\frac{d(\mathbf{p}, \widetilde{\mathbf{z}})^2 + 1}{d(\mathbf{p}, \widetilde{\mathbf{z}})^2 + \epsilon}\right)$$

  - Define activation of $\mathbf{p}$ on full embeddings $\mathbf{z}$ as maximum activation of its parts:

$$\text{act}(\mathbf{p}, \mathbf{z}) = \max_{\widetilde{\mathbf{z}} \in \text{parts}(\mathbf{z})} \text{act}(\mathbf{p}, \widetilde{\mathbf{z}})$$



Comparison between difference-of-logs and Gaussian of d:

$$\text{act}'(\mathbf{p}, \widetilde{\mathbf{z}}) = \exp\left(-\gamma \cdot d(\mathbf{p}, \widetilde{\mathbf{z}})^2\right)$$

In the plot $\epsilon = 0.001$, $\gamma = 1$

- How to measure **activation** of **part**-prototype $\mathbf{p} \in \mathbb{R}^{1\times1\times d}$ on a full embedding $\mathbf{z} \in \mathbb{R}^{w'\times h'\times d}$?

  - Break down $\mathbf{z}$ into all its **pieces** $\widetilde{\mathbf{z}}$ of size $1 \times 1 \times d$, denoted:

$$\text{parts}(\mathbf{z})$$

  - Measure L_2 **distance** between $\mathbf{p}$ and each part $\widetilde{\mathbf{z}}$ of $\mathbf{z}$:

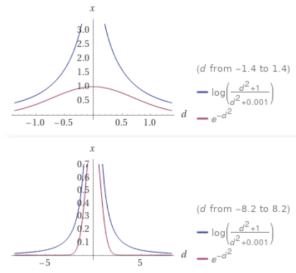
$$d(\mathbf{p}, \widetilde{\mathbf{z}}) = \|\mathbf{p} - \mathbf{z}\|$$

  - Convert distance into **activation**:

$$\text{act}(\mathbf{p}, \widetilde{\mathbf{z}}) = \log\left(\frac{d(\mathbf{p}, \widetilde{\mathbf{z}})^2 + 1}{d(\mathbf{p}, \widetilde{\mathbf{z}})^2 + \epsilon}\right)$$

  - Define activation of $\mathbf{p}$ on full embeddings $\mathbf{z}$ as maximum activation of its parts:

$$\text{act}(\mathbf{p}, \mathbf{z}) = \max_{\widetilde{\mathbf{z}} \in \text{parts}(\mathbf{z})} \text{act}(\mathbf{p}, \widetilde{\mathbf{z}})$$



Comparison between difference-of-logs and Gaussian of d:

$$\text{act}'(\mathbf{p}, \widetilde{\mathbf{z}}) = \exp\left(-\gamma \cdot d(\mathbf{p}, \widetilde{\mathbf{z}})^2\right)$$

In the plot $\epsilon = 0.001$, $\gamma = 1$

■ How to measure **activation** of **part**-prototype $\mathbf{p} \in \mathbb{R}^{1\times 1\times d}$ on a full embedding $\mathbf{z} \in \mathbb{R}^{w'\times h'\times d}$?

- Break down $\mathbf{z}$ into all its **pieces** $\widetilde{\mathbf{z}}$ of size $1 \times 1 \times d$, denoted:

$$\text{parts}(\mathbf{z})$$

- Measure L_2 **distance** between $\mathbf{p}$ and each part $\widetilde{\mathbf{z}}$ of $\mathbf{z}$:

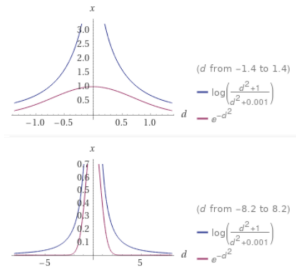
$$d(\mathbf{p}, \widetilde{\mathbf{z}}) = \|\mathbf{p} - \mathbf{z}\|$$

- Convert distance into **activation**:

$$\text{act}(\mathbf{p}, \widetilde{\mathbf{z}}) = \log\left(\frac{d(\mathbf{p}, \widetilde{\mathbf{z}})^2 + 1}{d(\mathbf{p}, \widetilde{\mathbf{z}})^2 + \epsilon}\right)$$

- Define activation of $\mathbf{p}$ on full embeddings $\mathbf{z}$ as maximum activation of its parts:

$$\text{act}(\mathbf{p}, \mathbf{z}) = \max_{\widetilde{\mathbf{z}} \in \text{parts}(\mathbf{z})} \text{act}(\mathbf{p}, \widetilde{\mathbf{z}})$$



Comparison between difference-of-logs and Gaussian of d:

$$\text{act}'(\mathbf{p}, \widetilde{\mathbf{z}}) = \exp\left(-\gamma \cdot d(\mathbf{p}, \widetilde{\mathbf{z}})^2\right)$$

In the plot $\epsilon = 0.001$, $\gamma = 1$

- How to measure **activation** of **part**-prototype $\mathbf{p} \in \mathbb{R}^{1\times1\times d}$ on a full embedding $\mathbf{z} \in \mathbb{R}^{w'\times h'\times d}$?

- Break down $\mathbf{z}$ into all its **pieces** $\widetilde{\mathbf{z}}$ of size $1 \times 1 \times d$, denoted:

$$\text{parts}(\mathbf{z})$$

- Measure L_2 **distance** between $\mathbf{p}$ and each part $\widetilde{\mathbf{z}}$ of $\mathbf{z}$:

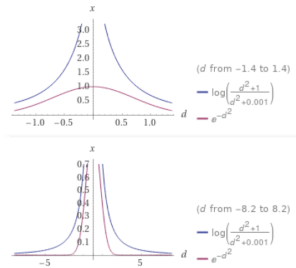
$$d(\mathbf{p}, \widetilde{\mathbf{z}}) = \|\mathbf{p} - \mathbf{z}\|$$

- Convert distance into **activation**:

$$\text{act}(\mathbf{p}, \widetilde{\mathbf{z}}) = \log\left(\frac{d(\mathbf{p}, \widetilde{\mathbf{z}})^2 + 1}{d(\mathbf{p}, \widetilde{\mathbf{z}})^2 + \epsilon}\right)$$

- Define activation of $\mathbf{p}$ on full embeddings $\mathbf{z}$ as maximum activation of its parts:

$$\text{act}(\mathbf{p}, \mathbf{z}) = \max_{\widetilde{\mathbf{z}} \in \text{parts}(\mathbf{z})} \text{act}(\mathbf{p}, \widetilde{\mathbf{z}})$$



Comparison between difference-of-logs and Gaussian of d:

$$\text{act}'(\mathbf{p}, \widetilde{\mathbf{z}}) = \exp\left(-\gamma \cdot d(\mathbf{p}, \widetilde{\mathbf{z}})^2\right)$$

In the plot $\epsilon = 0.001$, $\gamma = 1$

- How to measure **activation** of **part**-prototype $\mathbf{p} \in \mathbb{R}^{1\times1\times d}$ on a full embedding $\mathbf{z} \in \mathbb{R}^{w'\times h'\times d}$?

  - Break down $\mathbf{z}$ into all its **pieces** $\widetilde{\mathbf{z}}$ of size $1 \times 1 \times d$, denoted:

$$\text{parts}(\mathbf{z})$$

  - Measure L_2 **distance** between $\mathbf{p}$ and each part $\widetilde{\mathbf{z}}$ of $\mathbf{z}$:

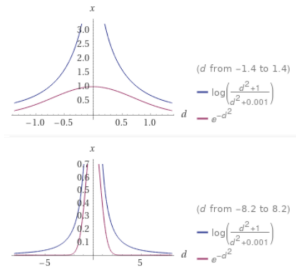
$$d(\mathbf{p}, \widetilde{\mathbf{z}}) = \|\mathbf{p} - \mathbf{z}\|$$

  - Convert distance into **activation**:

$$\text{act}(\mathbf{p}, \widetilde{\mathbf{z}}) = \log\left(\frac{d(\mathbf{p}, \widetilde{\mathbf{z}})^2 + 1}{d(\mathbf{p}, \widetilde{\mathbf{z}})^2 + \epsilon}\right)$$

  - Define activation of $\mathbf{p}$ on full embeddings $\mathbf{z}$ as maximum activation of its parts:

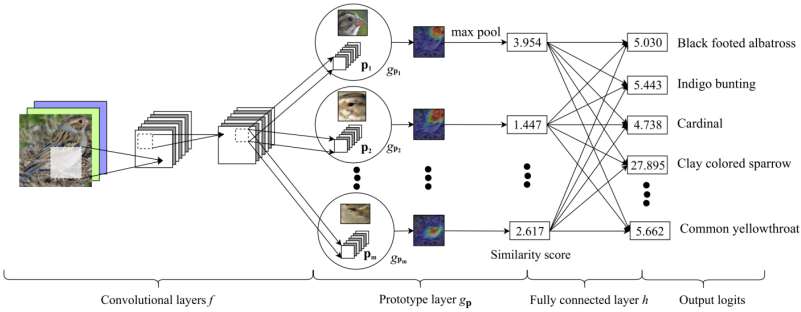
$$\text{act}(\mathbf{p}, \mathbf{z}) = \max_{\widetilde{\mathbf{z}} \in \text{parts}(\mathbf{z})} \text{act}(\mathbf{p}, \widetilde{\mathbf{z}})$$



Comparison between difference-of-logs and Gaussian of d:

$$\text{act}'(\mathbf{p}, \widetilde{\mathbf{z}}) = \exp\left(-\gamma \cdot d(\mathbf{p}, \widetilde{\mathbf{z}})^2\right)$$

In the plot $\epsilon = 0.001$, $\gamma = 1$



Remark:

- Convolutional filters slide over the input (first step from the left)
- Part-prototypes slide over the embeddings (second step from the left)

- How to ensure that part-prototypes are class-specific?

Desiderata:

- **Clustering**: *each* training example of class y should strongly activate *at least one* part-prototype $\mathbf{p}$ of that class.
  Can be converted into a regularization term:

$$\Omega_{cls} := \frac{1}{|S|} \sum_{(\mathbf{x},y)\in S} \min_{\mathbf{p}\in \mathrm{pps}_y} \min_{\widetilde{\mathbf{z}}\in \mathrm{parts}(f(\mathbf{x}))} \|\mathbf{p}-\widetilde{\mathbf{z}}\|^2$$

- **Separation**: *Every* training example of class y should activate *none* of the part-prototypes $\mathbf{p}$ of the other classes.
  Can be converted into a regularization term:

$$\Omega_{sep} := -\frac{1}{|S|} \sum_{(\mathbf{x},y)\in S} \min_{\mathbf{p}\notin \mathrm{pps}_y} \min_{\widetilde{\mathbf{z}}\in \mathrm{parts}(f(\mathbf{x}))} \|\mathbf{p}-\widetilde{\mathbf{z}}\|^2$$

- How to ensure that part-prototypes are class-specific?

Desiderata:

- **Clustering**: *each* training example of class y should strongly activate *at least one* part-prototype $\mathbf{p}$ of that class.
  Can be converted into a regularization term:

$$\Omega_{cls} := \frac{1}{|S|} \sum_{(\mathbf{x},y)\in S} \min_{\mathbf{p}\in \mathrm{pps}_y} \min_{\widetilde{\mathbf{z}}\in \mathrm{parts}(f(\mathbf{x}))} \|\mathbf{p}-\widetilde{\mathbf{z}}\|^2$$

- **Separation**: *Every* training example of class y should activate *none* of the part-prototypes $\mathbf{p}$ of the other classes.
  Can be converted into a regularization term:

$$\Omega_{sep} := -\frac{1}{|S|} \sum_{(\mathbf{x},y)\in S} \min_{\mathbf{p}\notin \mathrm{pps}_y} \min_{\widetilde{\mathbf{z}}\in \mathrm{parts}(f(\mathbf{x}))} \|\mathbf{p}-\widetilde{\mathbf{z}}\|^2$$

- How to ensure that part-prototypes are class-specific?

Desiderata:

- **Clustering**: *each* training example of class y should strongly activate *at least one* part-prototype $\mathbf{p}$ of that class.
  Can be converted into a regularization term:

$$\Omega_{cls} := \frac{1}{|S|} \sum_{(\mathbf{x},y)\in S} \min_{\mathbf{p}\in \mathrm{pps}_y} \min_{\widetilde{\mathbf{z}}\in \mathrm{parts}(f(\mathbf{x}))} \|\mathbf{p}-\widetilde{\mathbf{z}}\|^2$$

- **Separation**: *Every* training example of class y should activate *none* of the part-prototypes $\mathbf{p}$ of the other classes.
  Can be converted into a regularization term:

$$\Omega_{sep} := -\frac{1}{|S|} \sum_{(\mathbf{x},y)\in S} \min_{\mathbf{p}\notin \mathrm{pps}_y} \min_{\widetilde{\mathbf{z}}\in \mathrm{parts}(f(\mathbf{x}))} \|\mathbf{p}-\widetilde{\mathbf{z}}\|^2$$

- How to ensure that part-prototypes are class-specific?

Desiderata:

- **Clustering**: *each* training example of class y should strongly activate *at least one* part-prototype $\mathbf{p}$ of that class.
  Can be converted into a regularization term:

$$\Omega_{cls} := \frac{1}{|S|} \sum_{(\mathbf{x},y)\in S} \min_{\mathbf{p}\in \mathrm{pps}_y} \min_{\widetilde{\mathbf{z}}\in \mathrm{parts}(f(\mathbf{x}))} \|\mathbf{p}-\widetilde{\mathbf{z}}\|^2$$

- **Separation**: *Every* training example of class y should activate *none* of the part-prototypes $\mathbf{p}$ of the other classes.
  Can be converted into a regularization term:

$$\Omega_{sep} := -\frac{1}{|S|} \sum_{(\mathbf{x},y)\in S} \min_{\mathbf{p}\notin \mathrm{pps}_y} \min_{\widetilde{\mathbf{z}}\in \mathrm{parts}(f(\mathbf{x}))} \|\mathbf{p}-\widetilde{\mathbf{z}}\|^2$$

- How to ensure that part-prototypes are class-specific?

Desiderata:

- **Clustering**: *each* training example of class y should strongly activate *at least one* part-prototype $\mathbf{p}$ of that class.
  Can be converted into a regularization term:

$$\Omega_{cls} := \frac{1}{|S|} \sum_{(\mathbf{x},y)\in S} \min_{\mathbf{p}\in \mathrm{pps}_y} \min_{\widetilde{\mathbf{z}}\in \mathrm{parts}(f(\mathbf{x}))} \|\mathbf{p}-\widetilde{\mathbf{z}}\|^2$$

- **Separation**: *Every* training example of class y should activate *none* of the part-prototypes $\mathbf{p}$ of the other classes.
  Can be converted into a regularization term:

$$\Omega_{sep} := -\frac{1}{|S|} \sum_{(\mathbf{x},y)\in S} \min_{\mathbf{p}\notin \mathrm{pps}_y} \min_{\widetilde{\mathbf{z}}\in \mathrm{parts}(f(\mathbf{x}))} \|\mathbf{p}-\widetilde{\mathbf{z}}\|^2$$

- How to ensure that part-prototypes are interpretable?

Idea: "push" learned prototypes of class y to a concrete training example by solving:

$$\mathbf{p}_{\text{new}} \leftarrow \underset{\mathbf{p}_{\text{new}} \in \mathcal{Q}^{(y)}}{\operatorname{argmin}} \|\mathbf{p}_{\text{new}} - \mathbf{p}\|^2$$

where:

$$\mathcal{Q}^{(y)} = \{\widetilde{\mathbf{z}} \ : \ \widetilde{\mathbf{z}} \in \operatorname{parts}(f(\mathbf{x}_i)), \ y_i = y\}$$

is the set of all parts of (latent representations of) instances $\mathbf{x}_i$ in the prototype's class.

- Solved using SGD or similar.

- How to ensure that part-prototypes are interpretable?

Idea: "push" learned prototypes of class y to a concrete training example by solving:

$$\mathbf{p}_{\text{new}} \leftarrow \underset{\mathbf{p}_{\text{new}} \in \mathcal{Q}^{(y)}}{\text{argmin}} \|\mathbf{p}_{\text{new}} - \mathbf{p}\|^2$$

where:

$$\mathcal{Q}^{(y)} = \{\widetilde{\mathbf{z}} \ : \ \widetilde{\mathbf{z}} \in \text{parts}(f(\mathbf{x}_i)), \ y_i = y\}$$

is the set of all parts of (latent representations of) instances $\mathbf{x}_i$ in the prototype's class.

- Solved using SGD or similar.

- How to ensure that part-prototypes are interpretable?

Idea: "push" learned prototypes of class y to a concrete training example by solving:

$$\mathbf{p}_{\text{new}} \leftarrow \underset{\mathbf{p}_{\text{new}} \in \mathcal{Q}^{(y)}}{\text{argmin}} \|\mathbf{p}_{\text{new}} - \mathbf{p}\|^2$$

where:

$$\mathcal{Q}^{(y)} = \{\widetilde{\mathbf{z}} \ : \ \widetilde{\mathbf{z}} \in \text{parts}(f(\mathbf{x}_i)), \ y_i = y\}$$

is the set of all parts of (latent representations of) instances $\mathbf{x}_i$ in the prototype's class.

- Solved using SGD or similar.

- Training is split into three stages:
  - Load a pre-trained CNN and take its feature extractor $f(\mathbf{x})$, freeze the bottom layers.
  - Learn the part prototypes $\{\mathbf{p}\}$ of all classes while fine-tuning the top convolutional layers of f by minimizing:

$$\frac{1}{|S|} \sum_{(\mathbf{x},y)\in S} \ell_{\text{ce}}(\mathbf{x}, y) + \lambda_1 \Omega_{cls} + \lambda_2 \Omega sep$$

    At this stage, fix the weight vectors of the top dense layer to:

$$w_i^{(y)} = \begin{cases} 1 & \text{if } \mathbf{p}_i \text{ belongs to class } y \\ -\frac{1}{2} & \text{otherwise} \end{cases}$$

  - Periodically push prototypes close to training examples.
  - Once f and $\{\mathbf{p}\}$ are found, optimize weights of top dense layer W by optimizing the cross-entropy loss $\rightarrow$ convex problem

- Training is split into three stages:
  - Load a pre-trained CNN and take its feature extractor $f(\mathbf{x})$, freeze the bottom layers.
  - Learn the part prototypes $\{\mathbf{p}\}$ of all classes while fine-tuning the top convolutional layers of f by minimizing:

$$\frac{1}{|S|} \sum_{(\mathbf{x},y)\in S} \ell_{\text{ce}}(\mathbf{x}, y) + \lambda_1 \Omega_{cls} + \lambda_2 \Omega sep$$

    At this stage, fix the weight vectors of the top dense layer to:

$$w_i^{(y)} = \begin{cases} 1 & \text{if } \mathbf{p}_i \text{ belongs to class } y \\ -\frac{1}{2} & \text{otherwise} \end{cases}$$

  - Periodically push prototypes close to training examples.
  - Once f and $\{\mathbf{p}\}$ are found, optimize weights of top dense layer W by optimizing the cross-entropy loss $\rightarrow$ convex problem

- Training is split into three stages:
  - Load a pre-trained CNN and take its feature extractor $f(\mathbf{x})$, freeze the bottom layers.
  - Learn the part prototypes $\{\mathbf{p}\}$ of all classes while fine-tuning the top convolutional layers of f by minimizing:

$$\frac{1}{|S|} \sum_{(\mathbf{x},y)\in S} \ell_{\text{ce}}(\mathbf{x}, y) + \lambda_1 \Omega_{cls} + \lambda_2 \Omega sep$$

    At this stage, fix the weight vectors of the top dense layer to:

$$w_i^{(y)} = \begin{cases} 1 & \text{if } \mathbf{p}_i \text{ belongs to class } y \\ -\frac{1}{2} & \text{otherwise} \end{cases}$$

  - Periodically push prototypes close to training examples.
  - Once f and $\{\mathbf{p}\}$ are found, optimize weights of top dense layer W by optimizing the cross-entropy loss $\rightarrow$ convex problem

- Training is split into three stages:
  - Load a pre-trained CNN and take its feature extractor $f(\mathbf{x})$, freeze the bottom layers.
  - Learn the part prototypes $\{\mathbf{p}\}$ of all classes while fine-tuning the top convolutional layers of f by minimizing:

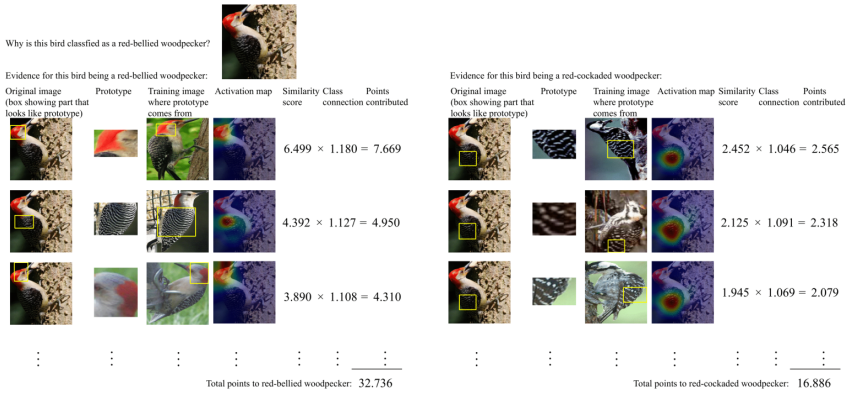
$$\frac{1}{|S|} \sum_{(\mathbf{x},y)\in S} \ell_{\text{ce}}(\mathbf{x}, y) + \lambda_1 \Omega_{cls} + \lambda_2 \Omega sep$$

    At this stage, fix the weight vectors of the top dense layer to:

$$w_i^{(y)} = \begin{cases} 1 & \text{if } \mathbf{p}_i \text{ belongs to class } y \\ -\frac{1}{2} & \text{otherwise} \end{cases}$$

  - Periodically push prototypes close to training examples.
  - Once f and $\{\mathbf{p}\}$ are found, optimize weights of top dense layer W by optimizing the cross-entropy loss $\rightarrow$ convex problem

Example



- Not quite counterfactual, but useful

Example

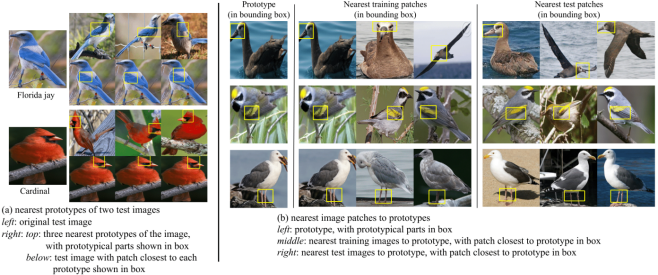


Figure 5: Nearest prototypes to images and nearest images to prototypes. The prototypes are learned from the training set.

- PPNets are the only method that explains where prototypes activate *and* where they come from!

Example

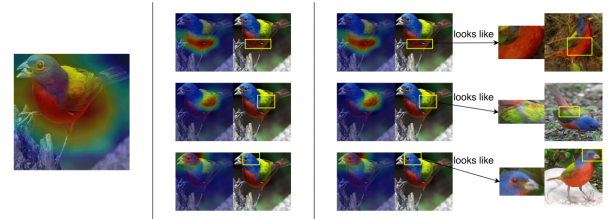


Figure 4: Visual comparison of different types of model interpretability: (a) object-level attention map (e.g., class activation map [56]); (b) part attention (provided by attention-based interpretable models); and (c) part attention with similar prototypical parts (provided by our model).

- Comparison between PPNets and other approaches to explainability

A Template

- Many concept-based models follow a **two-level structure**

- The model extracts a vector of **concept activations** from $\mathbf{x}$:

$$\mathbf{c}(\mathbf{x}) = (c_1(\mathbf{x}), \ldots, c_k(\mathbf{x})) \in \mathbb{R}^k$$

- Then it **aggregates** them into class scores, often in a simulatable [Lipton, 2018] manner, e.g., using a linear combination:

$$s_y(\mathbf{x}) := \langle \mathbf{w}^{(y)}(\mathbf{x}), \mathbf{c}(\mathbf{x}) \rangle = \sum_j w_j^{(y)}(\mathbf{x}) \cdot c_j(\mathbf{x})$$

where $\mathbf{w}^{(y)}(\mathbf{x}) \in \mathbb{R}^k$ is the weight vector associated to class y.

- Class probabilities are then obtained using a softmax: $P(y \mid \mathbf{x}) := \mathrm{softmax}(\mathbf{s}(\mathbf{x}))_y$.

- The concepts $\{c_j\}$ are:
  - Learned from data so to be discriminative and interpretable.
  - Black-box: what's "above" the concepts is interpretable, what's "underneath" is not.

A Template

- Many concept-based models follow a **two-level structure**

- The model extracts a vector of **concept activations** from $\mathbf{x}$:

$$\mathbf{c}(\mathbf{x}) = (c_1(\mathbf{x}), \ldots, c_k(\mathbf{x})) \in \mathbb{R}^k$$

- Then it **aggregates** them into class scores, often in a simulatable [Lipton, 2018] manner, e.g., using a linear combination:

$$s_y(\mathbf{x}) := \langle \mathbf{w}^{(y)}(\mathbf{x}), \mathbf{c}(\mathbf{x}) \rangle = \sum_j w_j^{(y)}(\mathbf{x}) \cdot c_j(\mathbf{x})$$

where $\mathbf{w}^{(y)}(\mathbf{x}) \in \mathbb{R}^k$ is the weight vector associated to class y.

- Class probabilities are then obtained using a softmax: $P(y \mid \mathbf{x}) := \mathrm{softmax}(\mathbf{s}(\mathbf{x}))_y$.

- The concepts $\{c_j\}$ are:
  - Learned from data so to be discriminative and interpretable.
  - Black-box: what's "above" the concepts is interpretable, what's "underneath" is not.

A Template

- Many concept-based models follow a **two-level structure**

- The model extracts a vector of **concept activations** from $\mathbf{x}$:

$$\mathbf{c}(\mathbf{x}) = (c_1(\mathbf{x}), \ldots, c_k(\mathbf{x})) \in \mathbb{R}^k$$

- Then it **aggregates** them into class scores, often in a simulatable [Lipton, 2018] manner, e.g., using a linear combination:

$$s_y(\mathbf{x}) := \langle \mathbf{w}^{(y)}(\mathbf{x}), \mathbf{c}(\mathbf{x}) \rangle = \sum_j w_j^{(y)}(\mathbf{x}) \cdot c_j(\mathbf{x})$$

where $\mathbf{w}^{(y)}(\mathbf{x}) \in \mathbb{R}^k$ is the weight vector associated to class y.

- Class probabilities are then obtained using a softmax: $P(y \mid \mathbf{x}) := \mathrm{softmax}(\mathbf{s}(\mathbf{x}))_y$.

- The concepts $\{c_j\}$ are:
  - Learned from data so to be discriminative and interpretable.
  - Black-box: what's "above" the concepts is interpretable, what's "underneath" is not.

A Template

- Many concept-based models follow a **two-level structure**

- The model extracts a vector of **concept activations** from $\mathbf{x}$:

$$\mathbf{c}(\mathbf{x}) = (c_1(\mathbf{x}), \ldots, c_k(\mathbf{x})) \in \mathbb{R}^k$$

- Then it **aggregates** them into class scores, often in a simulatable [Lipton, 2018] manner, e.g., using a linear combination:

$$s_y(\mathbf{x}) := \langle \mathbf{w}^{(y)}(\mathbf{x}), \mathbf{c}(\mathbf{x}) \rangle = \sum_j w_j^{(y)}(\mathbf{x}) \cdot c_j(\mathbf{x})$$

where $\mathbf{w}^{(y)}(\mathbf{x}) \in \mathbb{R}^k$ is the weight vector associated to class y.

- Class probabilities are then obtained using a softmax: $P(y \mid \mathbf{x}) := \mathrm{softmax}(\mathbf{s}(\mathbf{x}))_y$.

- The concepts $\{c_j\}$ are:
  - Learned from data so to be discriminative and interpretable.
  - Black-box: what's "above" the concepts is interpretable, what's "underneath" is not.

A Template

- Many concept-based models follow a **two-level structure**

- The model extracts a vector of **concept activations** from $\mathbf{x}$:

$$\mathbf{c}(\mathbf{x}) = (c_1(\mathbf{x}), \ldots, c_k(\mathbf{x})) \in \mathbb{R}^k$$

- Then it **aggregates** them into class scores, often in a simulatable [Lipton, 2018] manner, e.g., using a linear combination:

$$s_y(\mathbf{x}) := \langle \mathbf{w}^{(y)}(\mathbf{x}), \mathbf{c}(\mathbf{x}) \rangle = \sum_j w_j^{(y)}(\mathbf{x}) \cdot c_j(\mathbf{x})$$

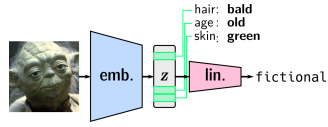
where $\mathbf{w}^{(y)}(\mathbf{x}) \in \mathbb{R}^k$ is the weight vector associated to class y.

- Class probabilities are then obtained using a softmax: $P(y \mid \mathbf{x}) := \mathrm{softmax}(\mathbf{s}(\mathbf{x}))_y$.

- The concepts $\{c_j\}$ are:
  - Learned from data so to be discriminative and interpretable.
  - Black-box: what's "above" the concepts is interpretable, what's "underneath" is not.

Key Feature: easy to extract a local explanation that captures how different concepts $\mathbf{c}$ contribute to a decision $(\mathbf{x}, y)$! These explanations take the form:

$$\text{expl}(\mathbf{x}, y) := \{(w_j^{(y)}(\mathbf{x}),\ c_j(\mathbf{x})) : j \in [k]\}$$



Remarks:

- The concepts and the weights are both integral to the explanation:
  - Concepts $\{c_j\}$ establish a vocabulary that enables communication with stakeholders
  - Weights $\{w_j(\mathbf{x})\}$ convey the relative importance of different concepts
- The prediction $y = f(\mathbf{x})$ is independent from $\mathbf{x}$ given the explanation $\text{expl}(\mathbf{x}, y)$ $\rightarrow$ the explanations is 100% **faithful** to the model's decision process.

Key Feature: easy to extract a local explanation that captures how different concepts $\mathbf{c}$ contribute to a decision $(\mathbf{x}, y)$! These explanations take the form:

$$\text{expl}(\mathbf{x}, y) := \{(w_j^{(y)}(\mathbf{x}),\ c_j(\mathbf{x})) : j \in [k]\}$$

Remarks:

- The concepts and the weights are both integral to the explanation:
  - Concepts $\{c_j\}$ establish a vocabulary that enables communication with stakeholders
  - Weights $\{w_j(\mathbf{x})\}$ convey the relative importance of different concepts
- The prediction $y = f(\mathbf{x})$ is independent from $\mathbf{x}$ given the explanation $\text{expl}(\mathbf{x}, y)$ $\rightarrow$ the explanations is 100% **faithful** to the model's decision process.

Key Feature: easy to extract a local explanation that captures how different concepts $\mathbf{c}$ contribute to a decision $(\mathbf{x}, y)$! These explanations take the form:

$$\text{expl}(\mathbf{x}, y) := \{(w_j^{(y)}(\mathbf{x}),\ c_j(\mathbf{x})) : j \in [k]\}$$

Remarks:

- The concepts and the weights are both integral to the explanation:
  - Concepts $\{c_j\}$ establish a vocabulary that enables communication with stakeholders
  - Weights $\{w_j(\mathbf{x})\}$ convey the relative importance of different concepts
- The prediction $y = f(\mathbf{x})$ is independent from $\mathbf{x}$ given the explanation $\text{expl}(\mathbf{x}, y) \rightarrow$ the explanations is 100% **faithful** to the model's decision process.

Take-away

- Concept-based models combine features of white and black-box models:
  - **Interpretability** (for parts of the prediction process)
  - **Faithfulness** of the produced explanations, they come for **free**
  - **High performance** on non-tabular data, thanks to representation learning

- **SENNs** upgrade linear models to representation learning; not 100% clear how to learn **interpretable concepts**

- Prototype and part-prototype models (partially) solve this issue by **mapping prototypes to examples** (or **parts** of examples)

- Still very much an **open area of research**! (Especially ensuring that concepts are interpretable)


Alvarez-Melis, D. and Jaakkola, T. S. (2018).
Towards robust interpretability with self-explaining neural networks.
In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pages 7786–7795.


Chen, C., Li, O., Tao, D., Barnett, A., Rudin, C., and Su, J. K. (2019).
This looks like that: Deep learning for interpretable image recognition.
Advances in Neural Information Processing Systems, 32:8930–8941.


Li, O., Liu, H., Chen, C., and Rudin, C. (2018).
Deep learning for case-based reasoning through prototypes: A neural network that explains its predictions.
In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32.


Lipton, Z. C. (2018).
The mythos of model interpretability: In machine learning, the concept of interpretability is both important and slippery.
Queue, 16(3):31–57.


Snell, J., Swersky, K., and Zemel, R. (2017).
Prototypical networks for few-shot learning.
In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 4080–4090.


Tibshirani, R. (1996).
Regression shrinkage and selection via the lasso.
Journal of the Royal Statistical Society: Series B (Methodological), 58(1):267–288.


Ustun, B. and Rudin, C. (2016).
Supersparse linear integer models for optimized medical scoring systems.
Machine Learning, 102(3):349–391.


Wu, M., Hughes, M. C., Parbhoo, S., Zazzi, M., Roth, V., and Doshi-Velez, F. (2018).
Beyond sparsity: Tree regularization of deep models for interpretability.
In *Thirty-Second AAAI Conference on Artificial Intelligence*.