Concept-based Models

Stefano Teso

Advanced Topics in Machine Learning & Optimization — 2023-24

The State of XAI

- **White-box models**:
  - Shallow decision trees, sparse linear models, rule lists, . . .
  - Explanations for free
  - Well suited for, e.g., tabular data
  - Low performance on non-tabular data, specifically because they don't support representation learning.

- **Black-box models** + **post-hoc explainers**:
  - Like neural nets + LIME, SHAP, Input Grads
  - High performance on non-tabular data like images and text thanks to representation learning.
  - Perturbation-based explanations are expensive to compute and can have high variance
  - Gradient-based explanations are widely applicable and cheap to compute, but they are often too "local".

Tree Regularization

- Regularizing black-boxes to be more transparent is a well-known strategy

■ Regularizing black-boxes to be more transparent is a well-known strategy

Idea: take a black-box f_θ and make it "more interpretable":

- 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*: "take in input data together with the parameters of the model and in reasonable time step through every calculation required to produce a prediction" Lipton (2018)

- Regularizing black-boxes to be more transparent is a well-known strategy

Idea: take a black-box f_θ and make it "more interpretable":

- 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*: "take in input data together with the parameters of the model and in reasonable time step through every calculation required to produce a prediction" Lipton (2018)

- Can we go **generalize** this strategy?

- Regularizing black-boxes to be more transparent is a well-known strategy

Idea: take a black-box f_θ and make it "more interpretable":

- 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*: "take in input data together with the parameters of the model and in reasonable time step through every calculation required to produce a prediction" Lipton (2018)

- Can we go **generalize** this strategy?

- 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 make it "more interpretable":

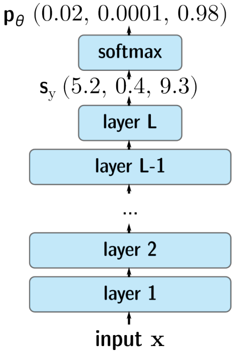
- If f_θ is a dense linear model, add a sparsifying L_1 regularizer so that its weight vector contains many zeros.
- This makes the model more *simulatable*: "take in input data together with the parameters of the model and in reasonable time step through every calculation required to produce a prediction" Lipton (2018)

- Can we go **generalize** this strategy?

- Can we make a neural network **behave like a decision tree** – so as to facilitate conversion into one using, e.g., LIME? **Yes** Wu et al. (2018)

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

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



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

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

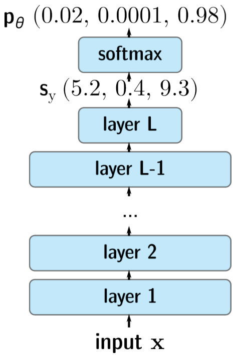
$$\frac{1}{|S|} \sum_{(\mathrm{x},y)\in S} -\log p_\theta(\mathrm{x}, y)$$

Tree Regularization

Instead of minimizing the usual loss, minimize the following augmented loss:

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

where $\Omega(\theta, \mathrm{x})$ is the **average depth** of a shallow DT that fits f_θ in the neighborhood of x



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

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

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

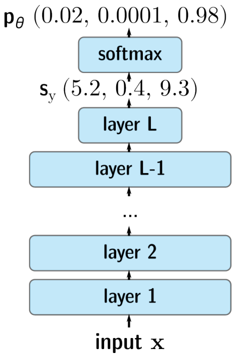
Tree Regularization

Instead of minimizing the usual loss, minimize the following augmented loss:

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

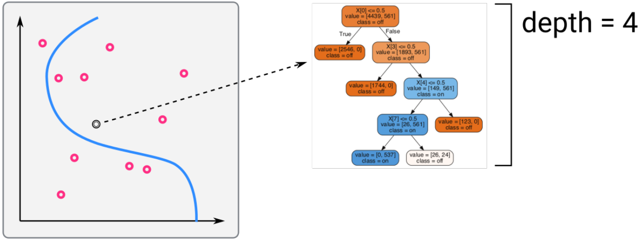
where $\Omega(\theta, \mathrm{x})$ is the **average depth** of a shallow DT that fits f_θ in the neighborhood of x

- Ω is small only if $f_\theta(\mathrm{x})$ can be simulated locally by a small DT



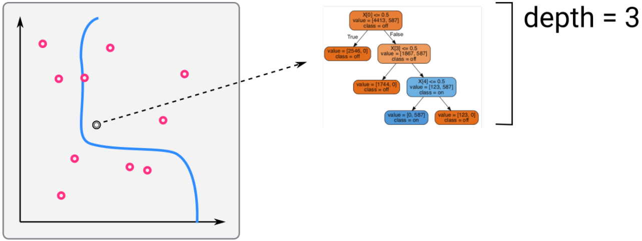
The structure of a typical feed-forward neural network with L layers.

Illustration



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

Illustration



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

- How to **compute** $\Omega(\theta, \theta)$?

- How to make **minimize** the augmented loss?

- How to **compute** $\Omega(\theta, \theta)$?

- How to make **minimize** the augmented loss?

Idea: learn an **auxiliary regressor** $d_\mu(\theta, \mathbf{x})$ that, given $\mathbf{x}$, **predicts the average depth of a DT** that fits f_θ **from the parameters** θ themselves

Tree Regularization Algorithm

- Fit f_θ on training set (cold start)

Tree Regularization Algorithm

- Fit f_θ on training set (cold start)
- Repeat:
    - Sample training instances $Q = \{\mathrm{x}_1, \ldots, \mathrm{x}_q\}$, either from the training set S or at random.

Tree Regularization Algorithm

- Fit f_θ on training set (cold start)
- Repeat:
  - Sample training instances $Q = \{\mathrm{x}_1, \ldots, \mathrm{x}_q\}$, either from the training set S or at random.
  - Label the examples in Q using the current model f_θ and **fit a decision tree** on them using `scikit-learn`. Then, $\Omega(\theta, \mathrm{x})$ is the average depth of the tree.

Tree Regularization Algorithm

- Fit f_θ on training set (cold start)
- Repeat:
  - Sample training instances $Q = \{\mathbf{x}_1, \ldots, \mathbf{x}_q\}$, either from the training set S or at random.
  - Label the examples in Q using the current model f_θ and **fit a decision tree** on them using `scikit-learn`. Then, $\Omega(\theta, \mathbf{x})$ is the average depth of the tree.
  - Fit $d_\mu(\theta, \mathbf{x})$ so that it approximates Ω:

$$\underset{\mu}{\operatorname{argmin}} \left(d_\mu(\theta, \mathbf{x}) - \Omega(\theta, \mathbf{x})\right)^2$$

Tree Regularization Algorithm

- Fit f_θ on training set (cold start)
- Repeat:
  - Sample training instances $Q = \{\mathrm{x}_1, \ldots, \mathrm{x}_q\}$, either from the training set S or at random.
  - Label the examples in Q using the current model f_θ and **fit a decision tree** on them using `scikit-learn`. Then, $\Omega(\theta, \mathrm{x})$ is the average depth of the tree.
  - Fit $d_\mu(\theta, \mathrm{x})$ so that it approximates Ω:

$$\operatorname*{argmin}_{\mu} \left(d_\mu(\theta, \mathrm{x}) - \Omega(\theta, \mathrm{x})\right)^2$$

  - Update θ by performing one *epoch* of gradient descent on the modified loss:

$$\operatorname*{argmin}_{\theta} \frac{1}{|S|} \sum_{(\mathrm{x},y)\in S} - \log p_\theta(\mathrm{x}, y) + \lambda \cdot \sum_{\mathrm{x}\in Q} d_\mu(\theta, \mathrm{x})$$

  The loss is now **fully differentiable**

Tree Regularization Algorithm

- Fit f_θ on training set (cold start)
- Repeat:
  - Sample training instances $Q = \{\mathrm{x}_1, \ldots, \mathrm{x}_q\}$, either from the training set S or at random.
  - Label the examples in Q using the current model f_θ and **fit a decision tree** on them using `scikit-learn`. Then, $\Omega(\theta, \mathrm{x})$ is the average depth of the tree.
  - Fit $d_\mu(\theta, \mathrm{x})$ so that it approximates Ω:

$$\operatorname*{argmin}_{\mu} \left(d_\mu(\theta, \mathrm{x}) - \Omega(\theta, \mathrm{x})\right)^2$$

  - Update θ by performing one *epoch* of gradient descent on the modified loss:

$$\operatorname*{argmin}_{\theta} \frac{1}{|S|} \sum_{(\mathrm{x},y)\in S} - \log p_\theta(\mathrm{x}, y) + \lambda \cdot \sum_{\mathrm{x}\in Q} d_\mu(\theta, \mathrm{x})$$

  The loss is now **fully differentiable**

- Dataset for training d_μ is $\{(\theta_k, \mathrm{x}_k), \Omega(\theta_k, \mathrm{x}_k) : k = 1, \ldots, s\}$ collected across epochs

Tree Regularization Algorithm

- Fit f_θ on training set (cold start)
- Repeat:
  - Sample training instances $Q = \{\mathbf{x}_1, \ldots, \mathbf{x}_q\}$, either from the training set S or at random.
  - Label the examples in Q using the current model f_θ and **fit a decision tree** on them using `scikit-learn`. Then, $\Omega(\theta, \mathbf{x})$ is the average depth of the tree.
  - Fit $d_\mu(\theta, \mathbf{x})$ so that it approximates Ω:

$$\operatorname*{argmin}_{\mu} \left(d_\mu(\theta, \mathbf{x}) - \Omega(\theta, \mathbf{x})\right)^2$$

  - Update θ by performing one *epoch* of gradient descent on the modified loss:

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

  The loss is now **fully differentiable**

- Dataset for training d_μ is $\{(\theta_k, \mathbf{x}_k), \Omega(\theta_k, \mathbf{x}_k) : k = 1, \ldots, s\}$ collected across epochs
- Makes sense as long as $|Q| \ll |S|$

Tree Regularization Algorithm

- Fit f_θ on training set (cold start)
- Repeat:
  - Sample training instances $Q = \{\mathbf{x}_1, \dots, \mathbf{x}_q\}$, either from the training set S or at random.
  - Label the examples in Q using the current model f_θ and **fit a decision tree** on them using `scikit-learn`. Then, $\Omega(\theta, \mathbf{x})$ is the average depth of the tree.
  - Fit $d_\mu(\theta, \mathbf{x})$ so that it approximates Ω:

$$\underset{\mu}{\operatorname{argmin}} \left(d_\mu(\theta, \mathbf{x}) - \Omega(\theta, \mathbf{x})\right)^2$$

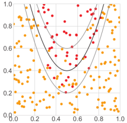
  - Update θ by performing one *epoch* of gradient descent on the modified loss:

$$\underset{\theta}{\operatorname{argmin}} \frac{1}{|S|} \sum_{(\mathbf{x},y)\in S} - \log p_\theta(\mathbf{x}, y) + \lambda \cdot \sum_{\mathbf{x}\in Q} d_\mu(\theta, \mathbf{x})$$

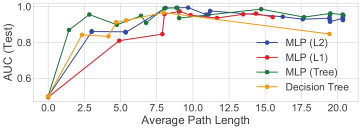
  The loss is now **fully differentiable**

- Dataset for training d_μ is $\{(\theta_k, \mathbf{x}_k), \Omega(\theta_k, \mathbf{x}_k) : k = 1, \dots, s\}$ collected across epochs
- Makes sense as long as $|Q| \ll |S|$
- Under the assumption that θ doesn't change "too much" across epochs, one can warm start training μ from the previous epoch.

Example: Fitting a Parabola

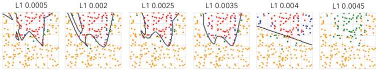


(a) Training Data and Binary Class Labels for 2D Parabola

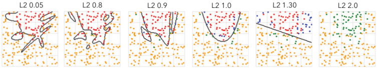


(b) Prediction quality and complexity as reg. strength λ varies

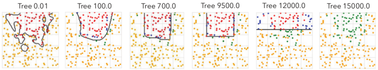
- For $\lambda = 9500$ (the exact value is not important) the tree-regularized network recovers exactly the shape of a **DT with depth** 2. Increasing λ further further flattens the tree to depth 1, at the cost of accuracy.



(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).

- Tree-based regularization has some limitations:
  - Training is either **computationally expensive**: must train d_μ in every epoch!

- Tree-based regularization has some limitations:
  - Training is either **computationally expensive**: must train d_μ in every epoch!
  - The regularizer is **approximate**: no guarantee that d_μ performs well, the depth prediction task is quite challenging!

- Tree-based regularization has some limitations:
  - Training is either **computationally expensive**: must train d_μ in every epoch!
  - The regularizer is **approximate**: no guarantee that d_μ performs well, the depth prediction task is quite challenging!
  - DTs only make sense for **tabular data**

- Tree-based regularization has some limitations:
  - Training is either **computationally expensive**: must train d_μ in every epoch!
  - The regularizer is **approximate**: no guarantee that d_μ performs well, the depth prediction task is quite challenging!
  - DTs only make sense for **tabular data**
  - $\rightarrow$ conflicts with **representation learning**

- Tree-based regularization has some limitations:
  - Training is either **computationally expensive**: must train d_μ in every epoch!
  - The regularizer is **approximate**: no guarantee that d_μ performs well, the depth prediction task is quite challenging!
  - DTs only make sense for **tabular data**
  - $\rightarrow$ conflicts with **representation learning**
- Can we **combine** the benefits of black-box and white-box models in a more direct and efficient manner?

- Tree-based regularization has some limitations:
  - Training is either **computationally expensive**: must train d_μ in every epoch!
  - The regularizer is **approximate**: no guarantee that d_μ performs well, the depth prediction task is quite challenging!
  - DTs only make sense for **tabular data**
  - $\rightarrow$ 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

From Inputs/Examples to Concepts

- Post-hoc explainers can be **unfaithful** (Teso, 2019)

Toward Faithful Explanatory Active Learning

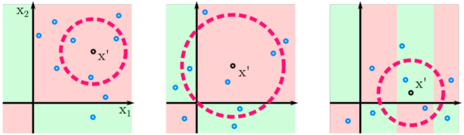


Fig. 1. Examples on which LIME produces unfaithful explanations. The decision surface of f is represented by the colored areas: light green means positive, light red negative. The pink circle represents the kernel k: points inside of it are assigned substantial weights while all others are not. Left: all synthetic examples with large weight have the same label. Middle: the synthetic examples fail to capture the non-additive interaction of the two features. Right: the kernel is too broad, and the synthetic dataset is highly complex and non-linear.

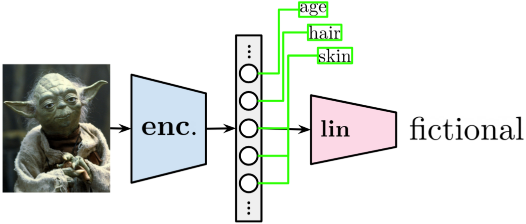
- Unfaithful explanations mean the user might be **correcting non-existent bugs!**

From Inputs/Examples to Concepts

- Low-level explanations are **ambiguous** (Rudin, 2019)

- Is it classified as “`positive`” because it is “`red`”, because it is “`sporty`”, or because it is a “`car`”?
- Human communication makes heavy use of **high-level concepts**!

Concept-based models (CBMs) are "gray-box" models [1] that support high-dimensional inputs & **representation learning** without giving up on **interpretability**.



(1) Map input to high-level **concepts** in a black-box manner

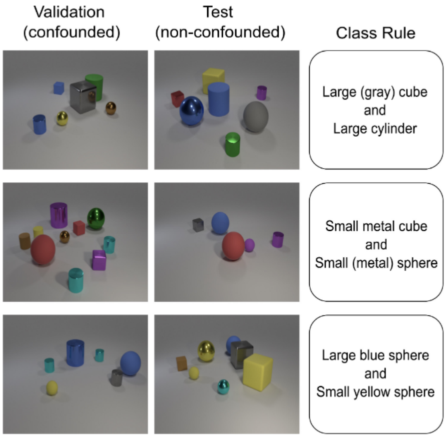
(2) Compute a prediction from the **concepts** in a white-box manner

(3) If concepts are interpretable, predictions admit **faithful** explanations like:

"`fictional`" because {**age** : -0.05, **hair** : $+0.02$, **skin** : 0.9}

The Simplest Case

- I will focus on **concepts** that are:
  - Concrete.
  - Categories: “`dog`”, “`ball`”.
  - Properties: “`red`”, “`shiny`”.
  - Atomic: no hierarchy, no real grammar.
  - Simple compositionality: “`red and shiny`”.
- This case is complicated enough already ;-)



(Source: The CLEVR-HANS data set (Stammer et al., 2021))

Concept-based Models (CBMs)

Concept-based Models

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)
  - Prototype-based Networks (ProtoNets, PCNs, PPNets)
  - Concept-bottleneck Models (CBNMs)

Self-explainable Neural Networks

A **linear model** has the form:

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

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** x taste good?

Consider a linear classifier:

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

Figure 1: A bunch of papaya fruits.

It is easy to read off what attributes are "for" and "against" x being tasty **for the model** – specifically because the model encodes independence assumptions, e.g., that the shape of 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**!

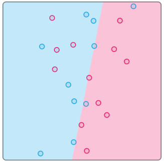


Illustration of a linear model. It cannot separate data with a complex, non-linear distribution.

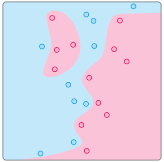


Illustration of a non-linear model. It works well with non-linear data like text, images, etc.

- **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**.

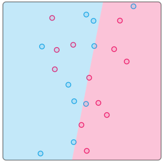


Illustration of a linear model. It cannot separate data with a complex, non-linear distribution.

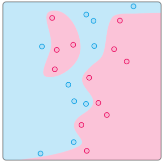


Illustration of a non-linear model. It works well with non-linear data like text, images, etc.

- **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 $\mathbf{x} \in \mathbb{R}^d$!

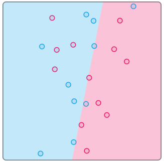


Illustration of a linear model. It cannot separate data with a complex, non-linear distribution.

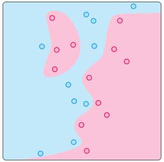
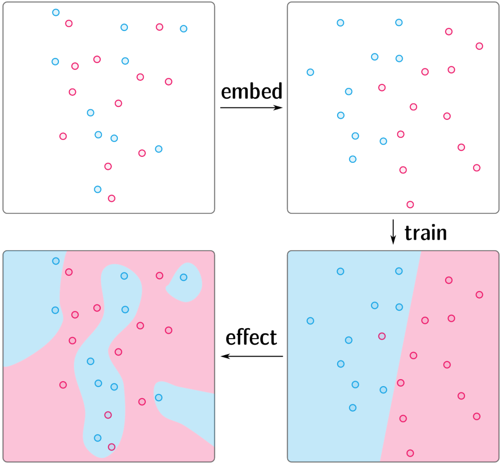


Illustration of a non-linear model. It works well with non-linear data like text, images, etc.

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

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. $\mathbf{x}$

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** $\mathbf{x} \in \mathbb{R}^d$

- 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:

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

- 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 $\mathbf{x} \in \mathbb{R}^d$**

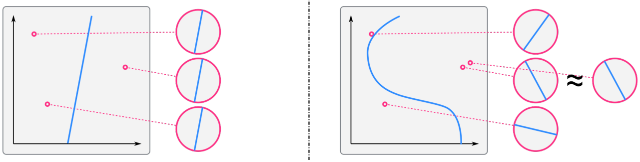
- Linear models associated to nearby inputs $\mathbf{x}$ **encouraged to be similar**, i.e., in the neighborhood of any $\mathbf{x}_0$ there exists a constant vector $\mathbf{w}_0$ that depends only on $\mathbf{x}_0$ and a "large enough" $\alpha > 0$ such that:

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

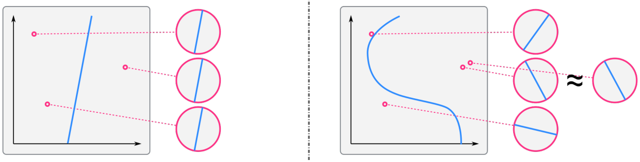
- If $\mathbf{w}(\mathbf{x}) \equiv \mathbf{w}$ is **constant** w.r.t. $\mathbf{x}$, we obtain a linear model again

- **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!



- **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**)

- 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:

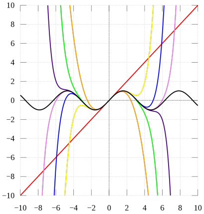
$$\mathbf{w}(\mathbf{x}) = \mathbf{w}(\mathbf{x}_0) + \underbrace{J(\mathbf{x}-\mathbf{x}_0)}_{\text{first-order term}} + \underbrace{\cdots}_{\text{quadratic+ terms}}$$

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

- 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.



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.

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

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 **local deviations from linearity**:

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

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

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 **local deviations from linearity**:

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

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.

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!

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.

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:

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

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**.

- 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.

- 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**.

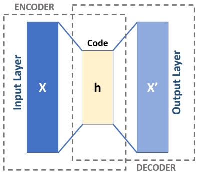
- An alternative useful for non-tabular data is to learn $\phi(\cdot)$ **automatically from the data beforehand**:

Example

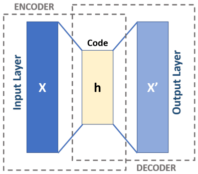
Train a convolutional neural network to classify ImageNet (1000 classes including many common objects) and then use the predictions made by the model to define 1000 different features, one for each class.

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

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



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

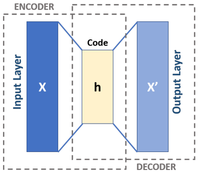


An **autoencoder** is defined as an encoder-decoder pair (ϕ, ψ):

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

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

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



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

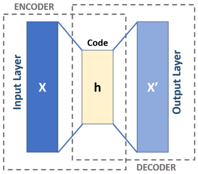
$$\phi : \mathbb{R}^d \to \mathbb{R}^k \qquad \psi : \mathbb{R}^k \to \mathbb{R}^d$$

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:

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

- 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:

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

- 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:

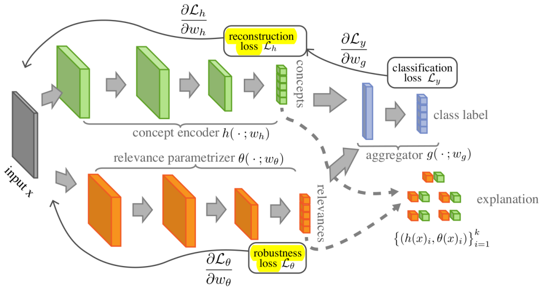


Figure 1: A SENN consists of three components: a **concept encoder** (green) that transforms the input into a small set of interpretable basis features; an **input-dependent parametrizer** (orange) that generates relevance scores; and an **aggregation function** that combines to produce a prediction. 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:

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

- Extra elements:
  - **Diversity**: encourage sparse concept activations by adding $\lambda'' \cdot \|\phi(\mathbf{x})\|_1$ to the loss

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**.

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:

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

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.

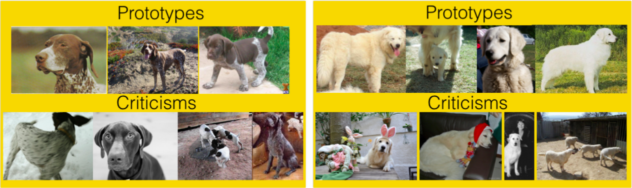
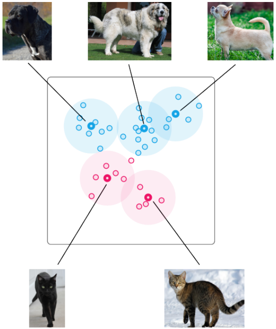


Figure 2: Learned prototypes and criticisms from Imagenet dataset (two types of dog breeds)

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)

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

Idea:

- Learn an **embedding function** $\phi : \mathbb{R}^d \rightarrow \mathbb{R}^c$

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 $\mathbf{c}^y := \frac{1}{S^y} \sum_{(\mathbf{x},k) \in S^y} \phi(\mathbf{x})$

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 $y \in \{1, \dots, v\}$ 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:

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

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

■ 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:

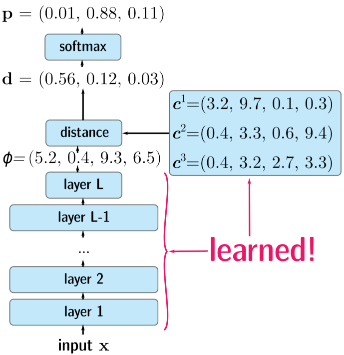
$$\mathbf{d} = (d(\phi(\mathbf{x}), \mathbf{c}^1), \ldots, d(\phi(\mathbf{x}), \mathbf{c}^v))$$

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

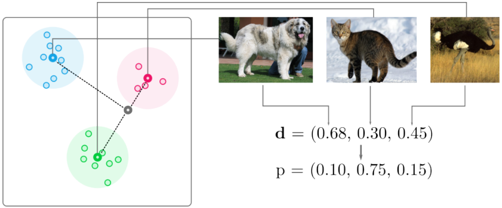
- Predicted **probability** of $\mathbf{x}$ belonging to class y proportional to **distance from prototype of that class**:

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

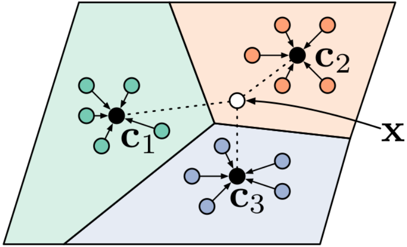
- Set of **all** parameters is $\theta = \{\phi, \mathbf{c}^1, \ldots, \mathbf{c}^v\}$.



- *Very* simple architecture



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



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

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

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

[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:

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

[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:

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

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]:

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

[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:

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

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

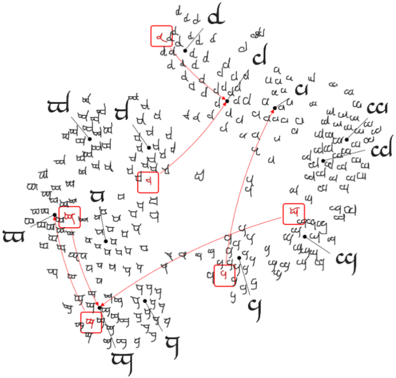


Figure 2: A t-SNE visualization of the embeddings learned by Prototypical networks on the Omniglot dataset. A subset of the Tengwar script is shown (an alphabet in the test set). Class prototypes are indicated in black. Several misclassified characters are highlighted in red along with arrows pointing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

- *Very* clear results

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

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

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

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.

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:

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

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.

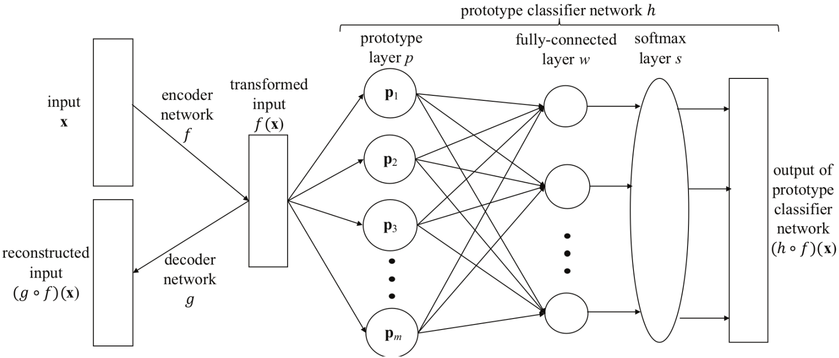
- **Prototype Layer** **[new!]**
  - Memorizes m prototypes $[\mathbf{p}_1, \ldots, \mathbf{p}_m]$, with $\mathbf{p}_j \in \mathbb{R}^q$
  - Outputs squared Euclidean distances between $f(\mathbf{x})$ and each prototype:

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

- **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**

- 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**

- 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

- 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:

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

- 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

- 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

- 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

- 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.

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.

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

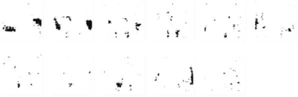


Figure 6: Decoded prototypes when we remove R_1 and R_2.

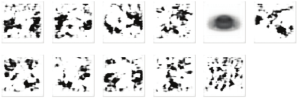


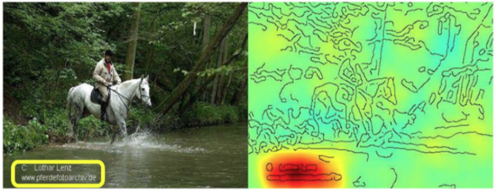
Figure 7: Decoded prototypes when we remove R_1.

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

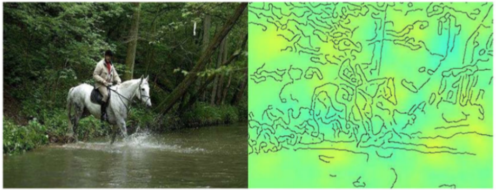
- Is autoencoding the way to go?

- 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

■ How would you describe why the image looks like a “clay colored sparrow”?

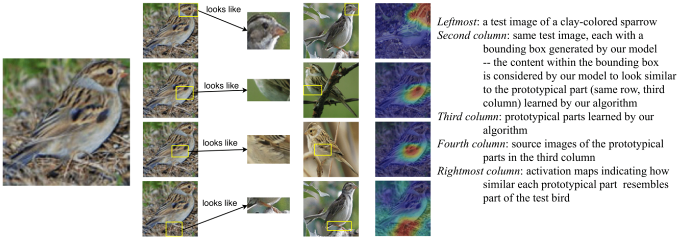


Figure 1: Image of a clay colored sparrow and how parts of it look like some learned prototypical parts of a clay colored sparrow used to classify the bird’s species.

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

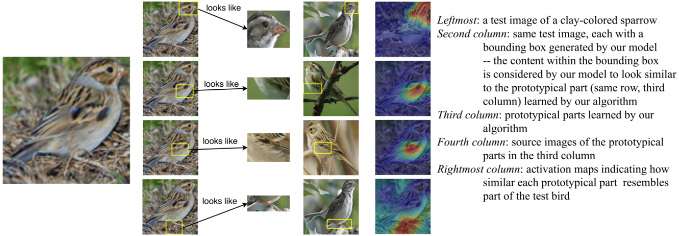


Figure 1: Image of a clay colored sparrow and how parts of it look like some learned prototypical parts of a clay colored sparrow used to classify the bird’s species.

- 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

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

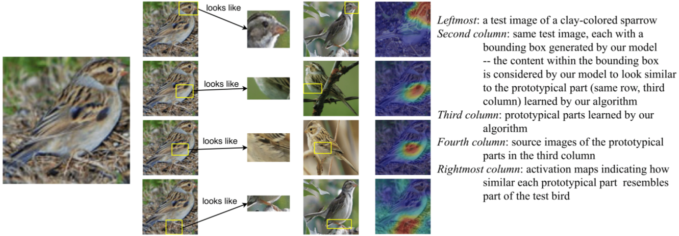
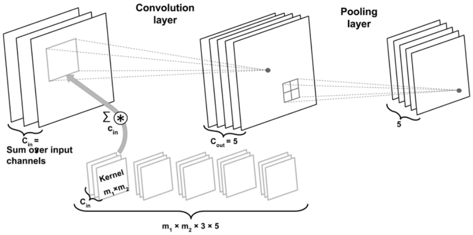


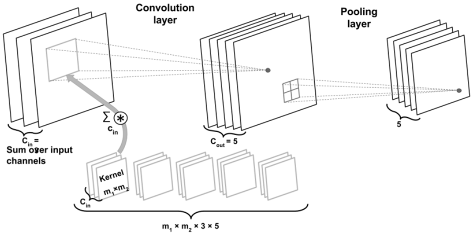
Figure 1: Image of a clay colored sparrow and how parts of it look like some learned prototypical parts of a clay colored sparrow used to classify the bird’s species.

- 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

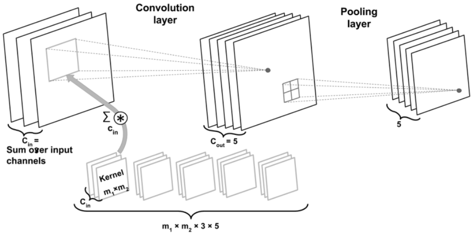
Idea: enable models to focus on parts of the image and compare them with prototypical parts of training images from a class – reasoning of the form “this looks like that”



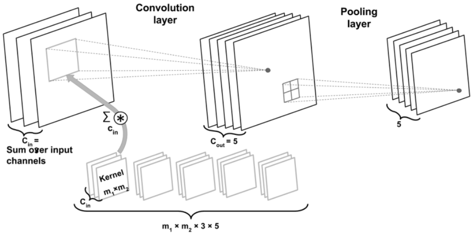
- **Structure**:
  - Given an **input** x of size $w \times h \times c$



- **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 $w' \times h' \times c$

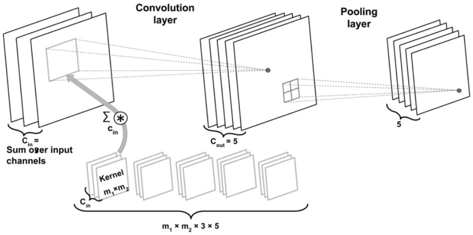


- **Structure**:
  - Given an **input** x of size $w \times h \times c$
  - A conv. layer has d **kernels** k_j, $j \in [d]$, each of size $w' \times h' \times c$
  - Each **kernel is convolved with the input** to obtain an output y_j of size $a \times b$, with $a = w - 2\left\lfloor \frac{w'}{2} \right\rfloor$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

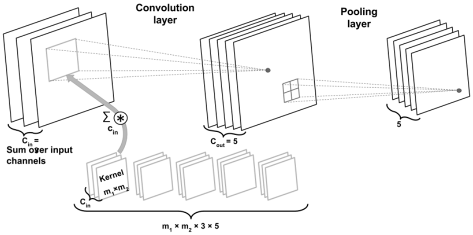


- **Structure**:
  - Given an **input** x of size $w \times h \times c$
  - A conv. layer has d **kernels** k_j, $j \in [d]$, each of size $w' \times h' \times c$
  - Each **kernel is convolved with the input** to obtain an output y_j of size $a \times b$, with $a = w - 2\left\lfloor \frac{w'}{2} \right\rfloor$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

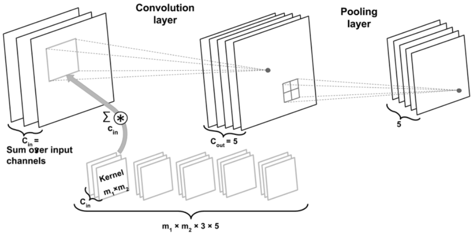
Refresher: Convolutional Networks



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



- 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



- 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 $h' \leq h$

- 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**

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

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

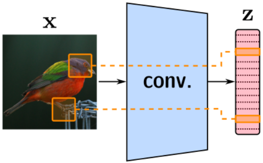
with $w' \leq w$ and $h' \leq h$

- 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**

ProtoPNets (Chen et al., 2019)

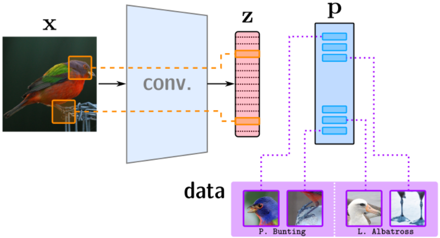
- **ProtoPNets** are a class of "gray-box" models that aim at achieveing high-performance through **representation learning** while providing **faithful** explanations for their own predictions.



- Embed input image $\mathbf{x}$ using convolutional layers → each "tube" in the latent representation can be mapped back to a region of the image.

ProtoPNets (Chen et al., 2019)

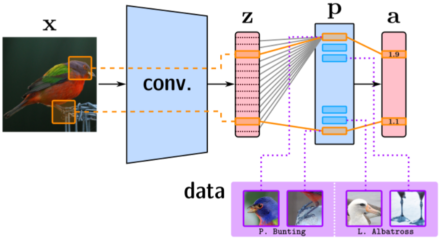
- **ProtoPNets** are a class of "gray-box" models that aim at achieveing high-performance through **representation learning** while providing **faithful** explanations for their own predictions.



- Jointly learn *k* **part-prototypes** for each class, i.e., "tubes" of latent representations of **parts of concrete training examples**. Learned using typical clustering losses (coverage & separation).

ProtoPNets (Chen et al., 2019)

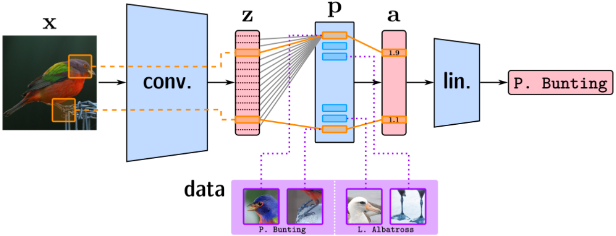
- **ProtoPNets** are a class of "gray-box" models that aim at achieveing high-performance through **representation learning** while providing **faithful** explanations for their own predictions.



- During inference, input "tubes" are compared to learned part-prototypes (also "tubes") and their similarity is computed - this captures similarity information between the input and training examples.

ProtoPNets (Chen et al., 2019)

- **ProtoPNets** are a class of "gray-box" models that aim at achieveing high-performance through **representation learning** while providing **faithful** explanations for their own predictions.



- The similarities themselves are passed through a linear layer and a `softmax` activation to obtain a distribution over classes.

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

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

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

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

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

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

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

  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-Prototype Networks

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

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

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

  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**]

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 $[\mathbf{p}_1, \dots, \mathbf{p}_m]$, 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 $\mathbf{z} = f(\mathbf{x})$:

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

[**it was squared L_2 distance**]

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 $[\mathbf{p}_1, \dots, \mathbf{p}_m]$, 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 $\mathbf{z} = f(\mathbf{x})$:

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

  [**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 $\mathbf{p} \in \mathbb{R}^{1\times 1\times d}$ on a full embedding $\mathbf{z} \in \mathbb{R}^{w'\times h'\times d}$?

- 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:

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

- 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:
    $$\mathrm{parts}(\mathbf{z})$$
  - Measure L_2 **distance** between $\mathbf{p}$ and each part $\widetilde{\mathbf{z}}$ of $\mathbf{z}$:
    $$d(\mathbf{p}, \widetilde{\mathbf{z}}) = \|\mathbf{p} - \mathbf{z}\|$$

- 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:
    $$\mathrm{parts}(\mathbf{z})$$
  - 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**:
    $$\mathrm{act}(\mathbf{p}, \widetilde{\mathbf{z}}) = \log\left(\frac{d(\mathbf{p}, \widetilde{\mathbf{z}})^2 + 1}{d(\mathbf{p}, \widetilde{\mathbf{z}})^2 + \epsilon}\right)$$

- 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:
  $$\mathrm{parts}(\mathbf{z})$$
  - 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**:
  $$\mathrm{act}(\mathbf{p}, \widetilde{\mathbf{z}}) = \log\left(\frac{d(\mathbf{p}, \widetilde{\mathbf{z}})^2 + 1}{d(\mathbf{p}, \widetilde{\mathbf{z}})^2 + \epsilon}\right)$$
  - Define activation of $\mathbf{p}$ on full embeddings $\mathbf{z}$ as maximum activation of its parts:
  $$\mathrm{act}(\mathbf{p}, \mathbf{z}) = \max_{\widetilde{\mathbf{z}} \in \mathrm{parts}(\mathbf{z})} \mathrm{act}(\mathbf{p}, \widetilde{\mathbf{z}})$$

■ 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 $\mathbf{z}$:

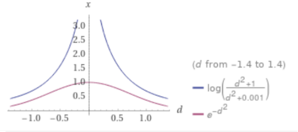
$$d(\mathbf{p}, \widetilde{\mathbf{z}}) = \|\mathbf{p} - \mathbf{z}\|$$

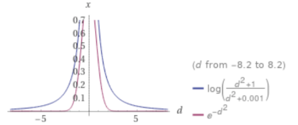
- Convert distance into **activation**:

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

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

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

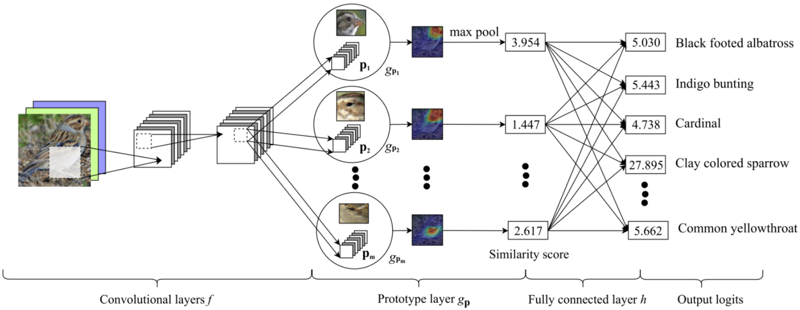




Comparison between difference-of-logs and Gaussian of d:

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

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?

■ 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.

■ 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:

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

■ 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.

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?

■ How to ensure that part-prototypes are interpretable?

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

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

where:

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

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

- 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:

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

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.

- 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:

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

- 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:

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

    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.

- 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:

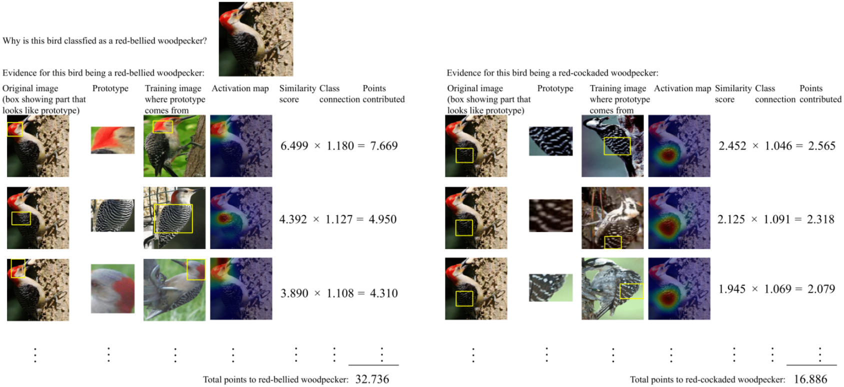
    $$\frac{1}{|S|}\sum_{(\mathbf{x},y)\in S} \ell_{\text{ce}}(\mathbf{x}, y) + \lambda_1 \Omega_{cls} + \lambda_2 \Omega sep$$

    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

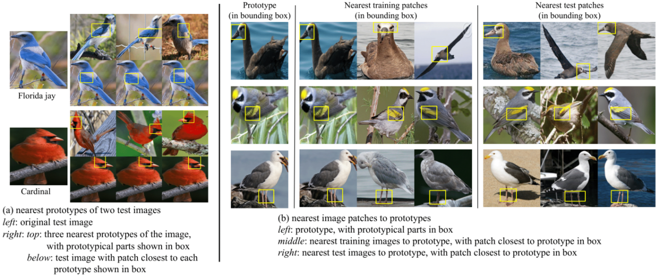


Figure 5: Nearest prototypes to images and nearest images to prototypes. The prototypes are learned from the training set.

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

Example

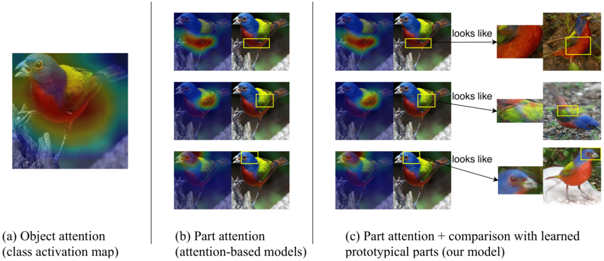


Figure 4: Visual comparison of different types of model interpretability: (a) object-level attention map (e.g., class activation map [56]); (b) part attention (provided by attention-based interpretable models); and (c) part attention with similar prototypical parts (provided by our model).

- Comparison between PPNets and other approaches to explainability

A Template

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

A Template

- Many concept-based models follow a **two-level structure**
- The model extracts a vector of **concept activations** from $\mathbf{x}$:

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

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.

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: $P(y \mid \mathbf{x}) := \mathrm{softmax}(\mathbf{s}(\mathbf{x}))_y$.

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

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

A Template

■ **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:

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

A Template

■ **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.

Easy!

- Faithful *by construction*
- Supplied *for free*
- No *hyperparameter tuning*

Easy!

- Faithful *by construction*
- Supplied *for free*
- No *hyperparameter tuning*

Useful!

A net mapping inputs to high-quality concepts can be used for a million things!

- Understanding, intervening, computing recourse, debugging
- Reasoning (**?**) & verification (**?**)!

Easy!

- Faithful *by construction*
- Supplied *for free*
- No *hyperparameter tuning*

Useful!

A net mapping inputs to high-quality concepts can be used for a million things!

- Understanding, intervening, computing recourse, debugging
- Reasoning (**?**) & verification (**?**)!

■ Am I Dreaming?

Easy!

- Faithful *by construction*
- Supplied *for free*
- No *hyperparameter tuning*

Useful!

A net mapping inputs to high-quality concepts can be used for a million things!

- Understanding, intervening, computing recourse, debugging
- Reasoning (**?**) & verification (**?**)!

■ Am I Dreaming? *Kinda :-(*

Easy!

- Faithful *by construction*
- Supplied *for free*
- No *hyperparameter tuning*

Useful!

A net mapping inputs to high-quality concepts can be used for a million things!

- Understanding, intervening, computing recourse, debugging
- Reasoning (**?**) & verification (**?**)!

- Am I Dreaming? *Kinda :-(*
- *Interpretability not well defined!*
- Each CBM implements it in its own way:
  - Activation sparsity
  - Orthonormality
  - Similarity to concrete examples
  - ... all *unsupervised* approaches

Concept Bottleneck Models

Pang Wei Koh [*1] **Thao Nguyen** [*1 2] **Yew Siang Tang** [*1]
Stephen Mussmann [1] **Emma Pierson** [1] **Been Kim** [2] **Percy Liang** [1]

Abstract

We seek to learn models that we can interact with using high-level concepts: if the model did not think there was a bone spur in the x-ray, would it still predict severe arthritis? State-of-the-art models today do not typically support the manipulation of concepts like “the existence of bone spurs”, as they are trained end-to-end to go directly from raw input (e.g., pixels) to output (e.g., arthritis severity). We revisit the classic idea of first predicting concepts that are provided at training time, and then using these concepts to predict the label. By construction, we can intervene on these *concept bottleneck models* by editing their predicted concept values and propagating these changes to the final prediction. On x-ray grading and bird identification, concept bottleneck models achieve competitive accuracy with standard end-to-end models, while enabling interpretation in terms of high-level clinical concepts (“bone spurs”) or bird attributes (“wing color”). These

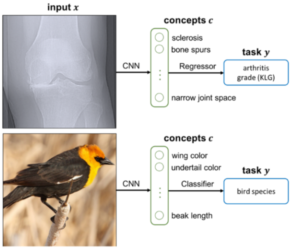


Figure 1. We study concept bottleneck models that first predict an intermediate set of human-specified concepts c, then use c to predict the final output y. We illustrate the two applications we consider: knee x-ray grading and bird identification.

(Source: (Koh et al., 2020) Also: Concept Whitening (Chen et al., 2020))

Concept Annotations Are Not Enough!

DO CONCEPT BOTTLENECK MODELS LEARN AS INTENDED?

Andrei Margeloiu*
University of Cambridge
am2770@cam.ac.uk

Matthew Ashman*
University of Cambridge
mca39@cam.ac.uk

Umang Bhatt*
University of Cambridge
usb20@cam.ac.uk

Yanzhi Chen
University of Edinburgh

Mateja Jamnik
University of Cambridge

Adrian Weller
University of Cambridge
The Alan Turing Institute

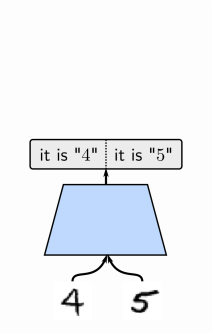
ABSTRACT

Concept bottleneck models map from raw inputs to concepts, and then from concepts to targets. Such models aim to incorporate pre-specified, high-level concepts into the learning procedure, and have been motivated to meet three desiderata: interpretability, predictability, and intervenability. However, we find that concept bottleneck models struggle to meet these goals. Using post hoc interpretability methods, we demonstrate that concepts do not correspond to anything semantically meaningful in input space, thus calling into question the usefulness of concept bottleneck models in their current form.

(Source: (**??**))

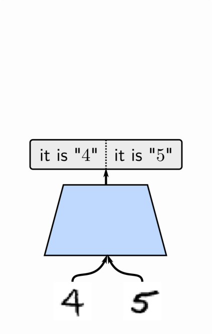
Concept Leakage = Unintended Semantics

① Fit two concepts to recognize `MNIST` images of "4"s and "5"s using **full concept annotations**

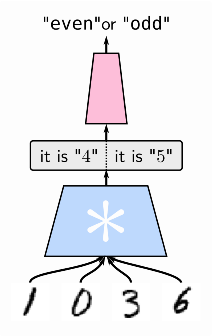


Concept Leakage = Unintended Semantics

① Fit two concepts to recognize MNIST images of "4"s and "5"s using **full concept annotations**



② Use the learned concepts, predict parity of **remaining digits** (i.e., all except "4" and "5")



- Performance is **much better than random**!

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)

References

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.

Chen, Z., Bei, Y., and Rudin, C. (2020). Concept whitening for interpretable image recognition. *Nature Machine Intelligence*, 2(12):772–782.

Koh, P. W., Nguyen, T., Tang, Y. S., Mussmann, S., Pierson, E., Kim, B., and Liang, P. (2020). Concept bottleneck models. In *International Conference on Machine Learning*, pages 5338–5348. PMLR.

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.

Rudin, C. (2019). Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nature Machine Intelligence*, 1(5):206–215.

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.

Stammer, W., Schramowski, P., and Kersting, K. (2021). Right for the right concept: Revising neuro-symbolic concepts by interacting with their explanations. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 3619–3629.

Teso, S. (2019). Toward faithful explanatory active learning with self-explainable neural nets. In *Proceedings of the Workshop on Interactive Adaptive Learning (IAL 2019)*, pages 4–16.

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*.