Regularization

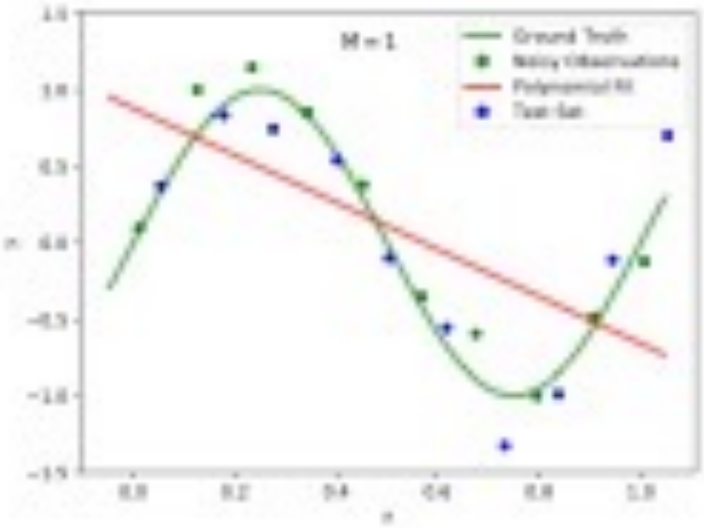
Deep Learning & Applications

Regularization

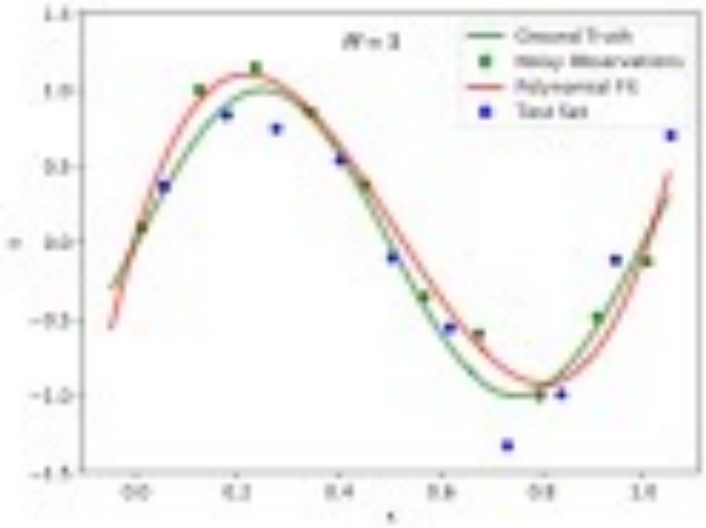
- Parameter Penalties
- Early Stopping
- Ensemble models
- Dropouts
- Data Augmentation

L1/L2 Regularization

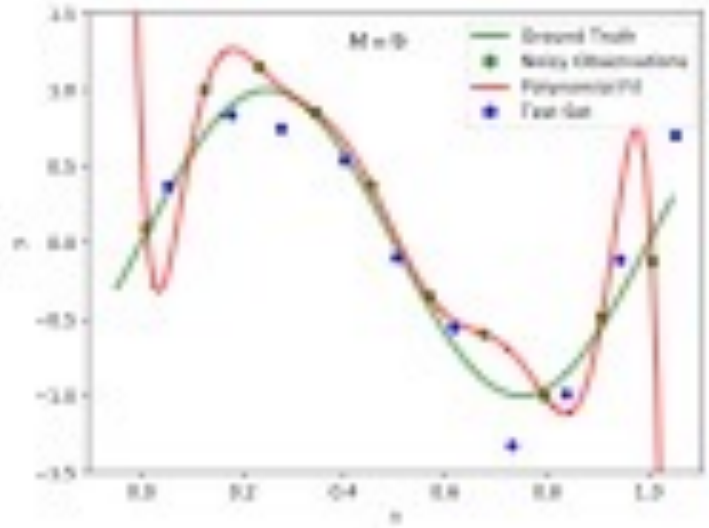
Recap: Capacity, Overfitting and Underfitting



Capacity too low



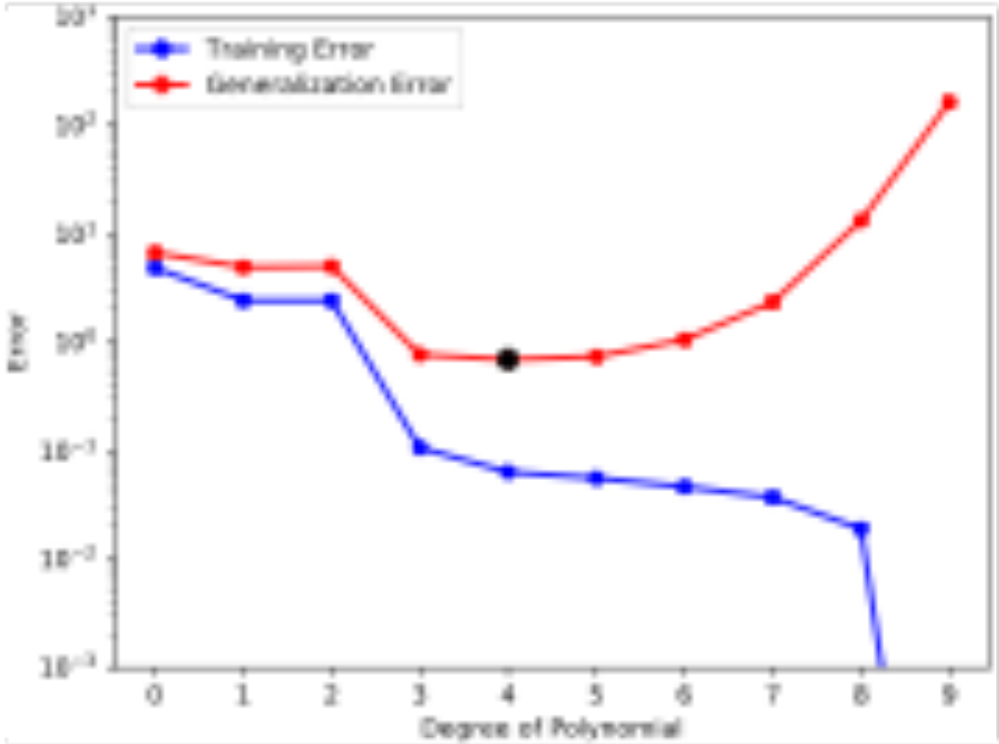
Capacity about right

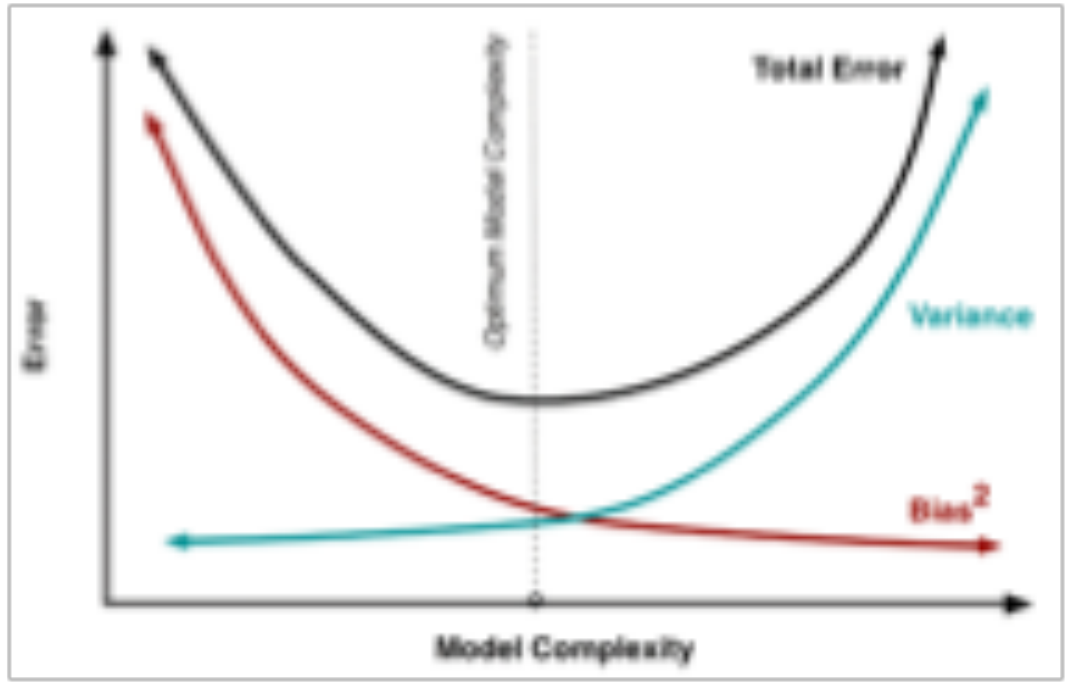


Capacity too high

- **Underfitting:** Model too simple, does not achieve low error on training set
- **Overfitting:** Training error small, but test error (= generalization error) large
- **Regularization:** Take model from third regime (right) to second regime (middle)

Recap: Capacity, Overfitting and Underfitting





Regularization:

- Trades **increased bias** for **reduced variance**
- Goal is to **minimize generalization error** despite using **large model family**

Regularization / Regularizing Effects

Goal: reduce overfitting

usually achieved by reducing model capacity and/or reduction of the variance of the predictions (as explained last lecture)

- L_1-regularization => LASSO regression
- L_2-regularization => Ridge regression (Thikonov regularization)

Basically, a "weight shrinkage" or a "penalty against complexity"

Logistic Regression Hyperparameters

Regular loss function to minimize during training

$$L(\mathbf{w}, b \mid \mathbf{x}) = -\sum_{i=1} \left[y^{(i)} \log\left(\sigma\left(z^{(i)}\right)\right) + \left(1 - y^{(i)}\right) \log\left(1 - \sigma\left(z^{(i)}\right)\right) \right]$$

L1 Norm /
LASSO (Least Absolute Shrinkage and Selection Operator)

L1 norm: $\lambda \, || \, \mathbf{w} \, ||_1 = \lambda \sum_{j=1}^{m} | \, w_j \, |$

L1 Regularization / LASSO (Embedded)

Least Absolute Shrinkage and Selection Operator

L1 penalty against complexity

$$\lambda \, || \, \mathbf{w} \, ||_1 = \lambda \sum_{j=1}^{m} | \, w_j \, |$$

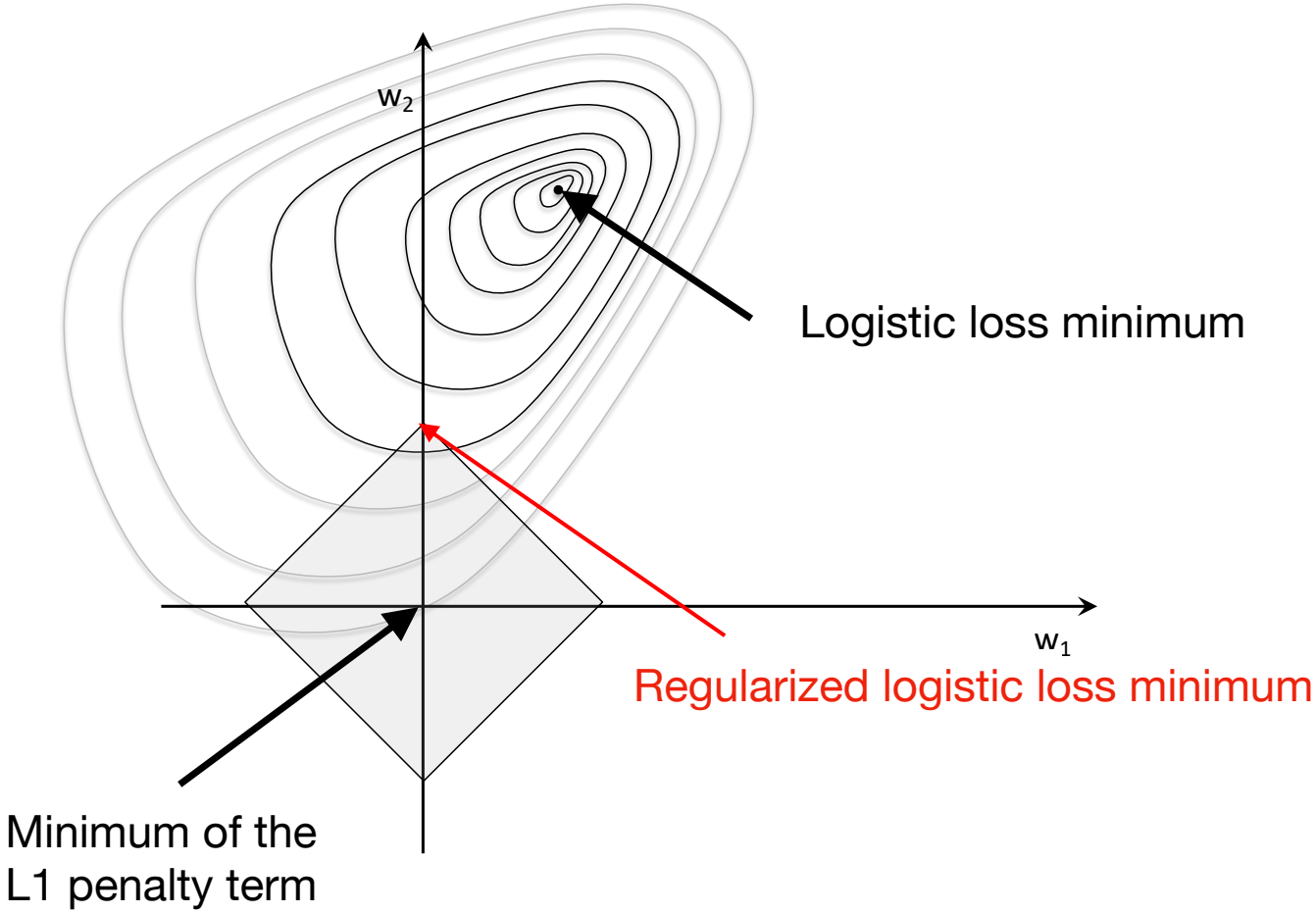
hyperparameter

L1-penalized loss

$$L_{L1}(\mathbf{w}, b \mid \mathbf{x}) = -\sum_{i=1} \left[y^{(i)} \log \left(\sigma \left(z^{(i)} \right) \right) + \left(1 - y^{(i)} \right) \log \left(1 - \sigma \left(z^{(i)} \right) \right) \right] + \lambda \| w \|_1$$

L1 Regularization / LASSO (Embedded)

Least Absolute Shrinkage and Selection Operator



L_2 Regularization for Linear Models (e.g., Logistic Regression)

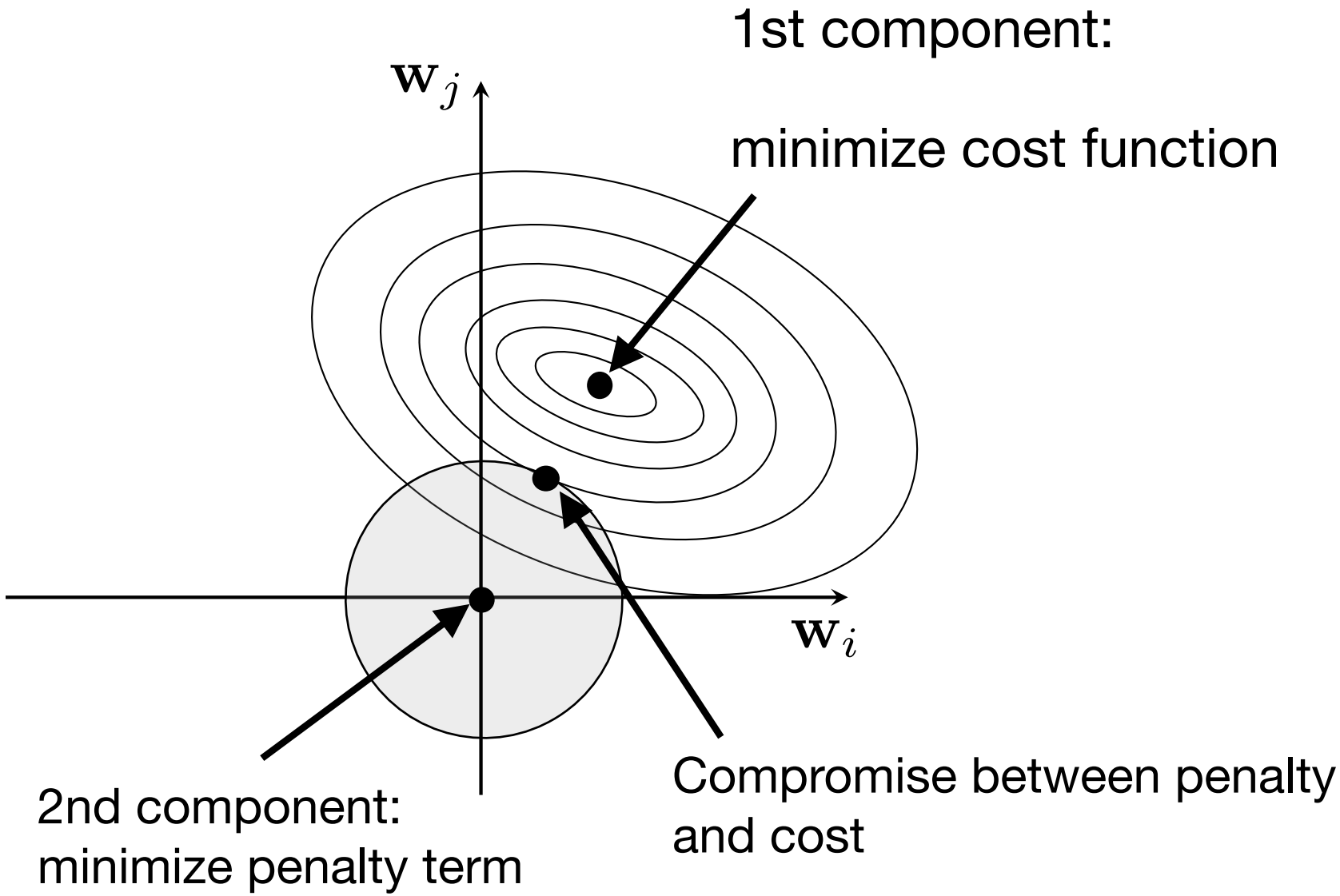
$$\text{Cost}_{\mathbf{w},\mathbf{b}} = \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}(y^{[i]}, \hat{y}^{[i]})$$

$$\text{L2-Regularized-Cost}_{\mathbf{w},\mathbf{b}} = \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}(y^{[i]}, \hat{y}^{[i]}) + \frac{\lambda}{n}\sum_{j} w_j^2$$

where: $\sum_j w_j^2 = ||\mathbf{w}||_2^2$

and λ is a hyperparameter

Geometric Interpretation of L_2 Regularization



L_2 Regularization for Multilayer Neural Networks

$$\text{L2-Regularized-Cost}_{\mathbf{w},\mathbf{b}} = \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}(y^{[i]}, \hat{y}^{[i]}) + \frac{\lambda}{n}\sum_{l=1}^{L}||\mathbf{w}^{(l)}||_F^2$$

sum over layers

where $||\mathbf{w}^{(l)}||_F^2$ is the Frobenius norm (squared):

$$||\mathbf{w}^{(l)}||_F^2 = \sum_i \sum_j (w_{i,j}^{(l)})^2$$

L_2 Regularization for Neural Nets

Regular gradient descent update:

$$w_{i,j} := w_{i,j} - \eta \frac{\partial \mathcal{L}}{\partial w_{i,j}}$$

Gradient descent update with L2 regularization:

$$w_{i,j} := w_{i,j} - \eta \left(\frac{\partial \mathcal{L}}{\partial w_{i,j}} + \frac{2\lambda}{n} w_{i,j} \right)$$

L2 Regularization

Weight decay (=ridge regression) uses a (squared) L_2 penalty $\mathcal{R}(\mathbf{w}) = \frac{1}{2}\|\mathbf{w}\|_2^2$:

$$\begin{aligned}\tilde{\mathcal{L}}(\mathcal{X}, \mathbf{w}) &= \mathcal{L}(\mathcal{X}, \mathbf{w}) + \lambda\ \mathcal{R}(\mathbf{w}) \\ &= \mathcal{L}(\mathcal{X}, \mathbf{w}) + \frac{\lambda}{2}\mathbf{w}^\top\mathbf{w}\end{aligned}$$

The **parameter updates** during gradient descent are given by:

$$\begin{aligned}\mathbf{w}^{t+1} &= \mathbf{w}^t - \eta\,\nabla_{\mathbf{w}}\tilde{\mathcal{L}}(\mathcal{X}, \mathbf{w}^t) \\ &= \mathbf{w}^t - \eta\,\left(\nabla_{\mathbf{w}}\mathcal{L}(\mathcal{X}, \mathbf{w}^t) + \lambda\ \mathbf{w}^t\right) \\ &= (1 - \eta\,\lambda)\ \mathbf{w}^t - \eta\,\nabla_{\mathbf{w}}\mathcal{L}(\mathcal{X}, \mathbf{w}^t)\end{aligned}$$

Thus, we **decay the weights** at each training iteration before the gradient update.

L2 vs. L1 Regularization

Example: Assume 3 input features: $\mathbf{x} = (1, 2, 1)^\top$

The following two **linear classifiers** $f_{\mathbf{w}}(\mathbf{x}) = \sigma(\mathbf{w}^\top \mathbf{x})$ yield the **same result/loss:**

- $\mathbf{w}_1 = (0, 0.75, 0)^\top \Rightarrow$ ignores 2 features
- $\mathbf{w}_2 = (0.25, 0.5, 0.25)^\top \Rightarrow$ takes all features into account

But the L1 and L2 regularizer **prefer different solutions!**

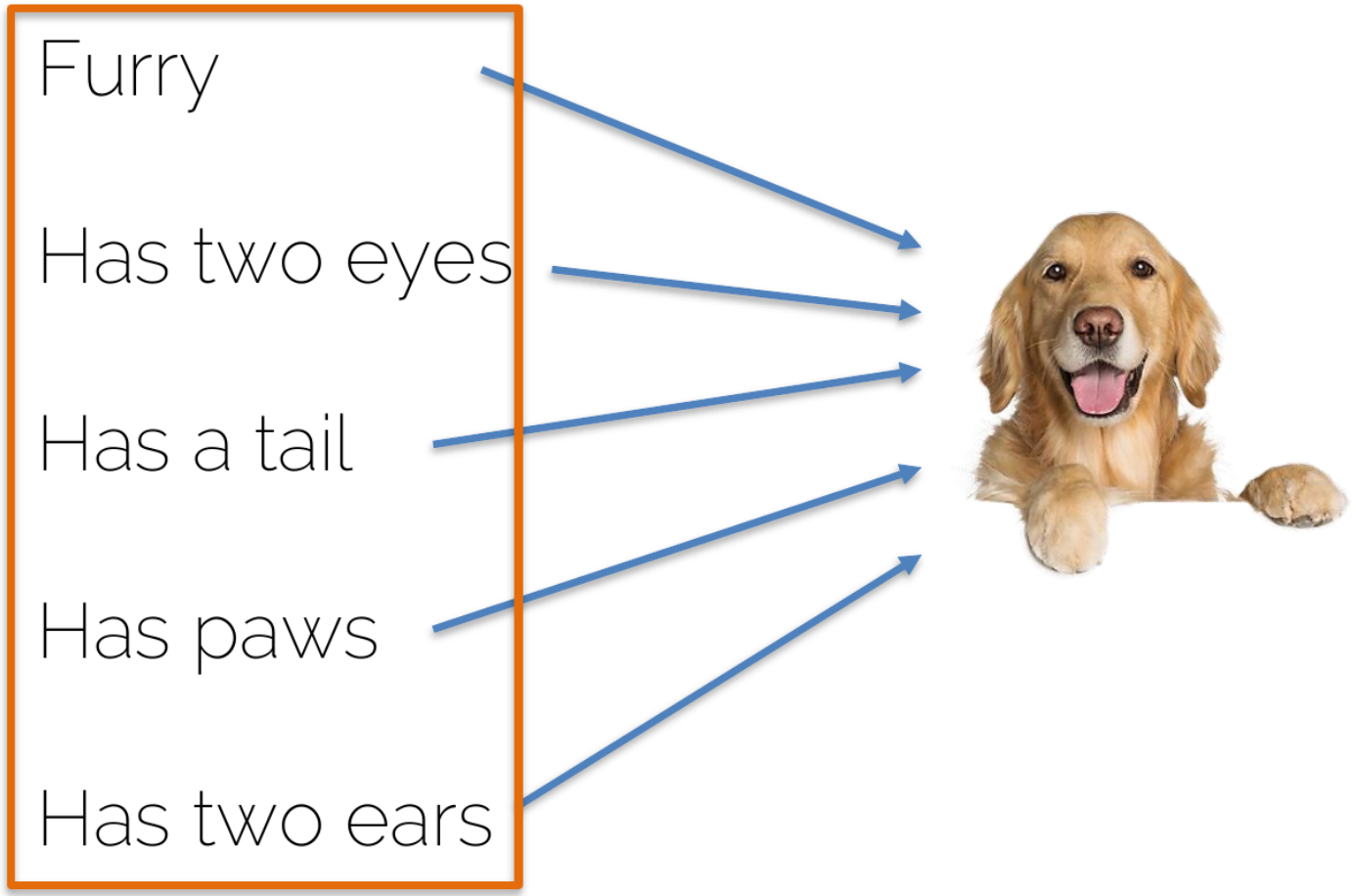
L2 Regularization:

- $\|\mathbf{w}_1\|_2^2 = 0 + 0.75^2 + 0 = 0.5625$
- $\|\mathbf{w}_2\|_2^2 = 0.25^2 + 0.5^2 + 0.25^2 = \mathbf{0.375}$

L1 Regularization:

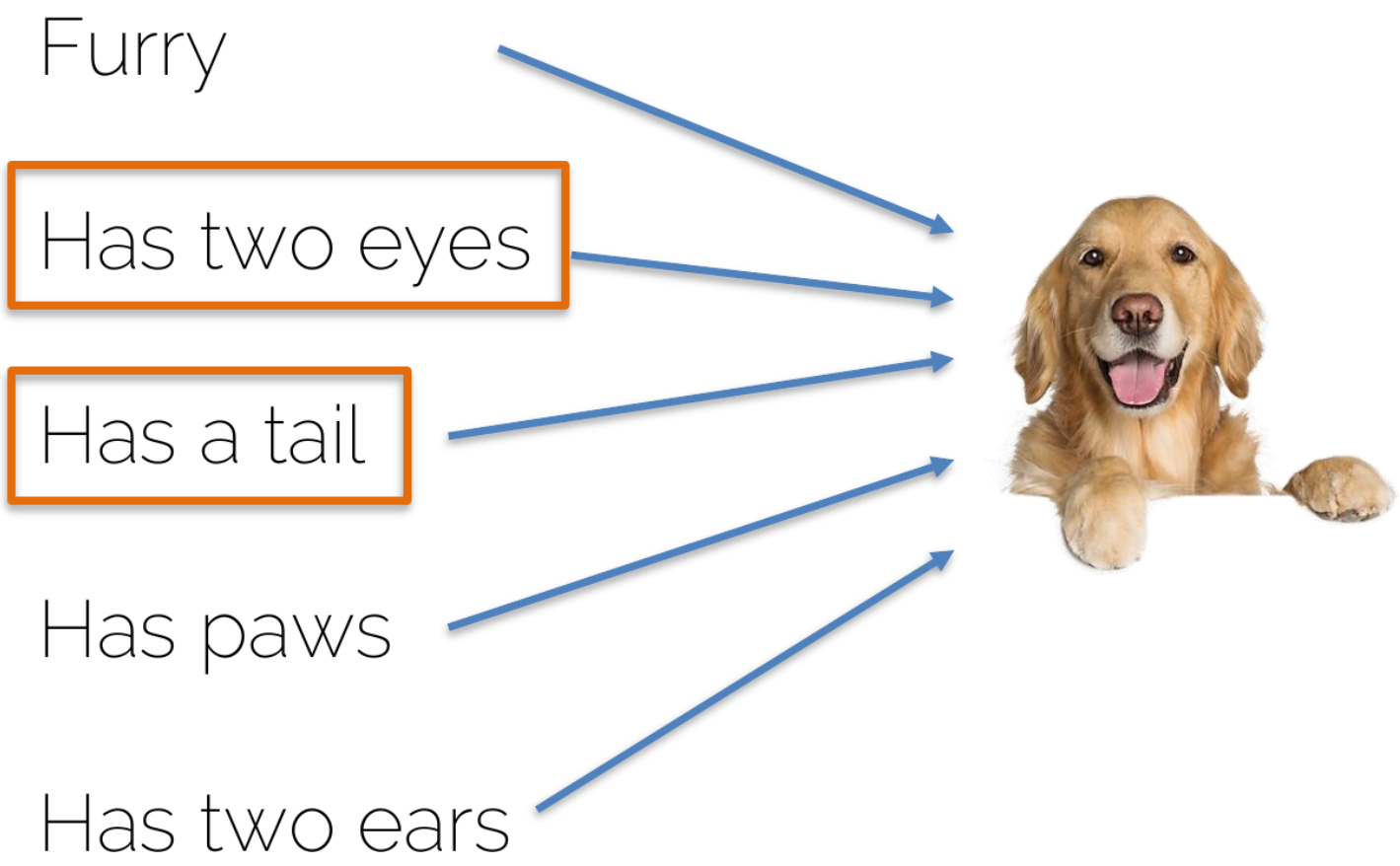
- $\|\mathbf{w}_1\|_1 = 0 + 0.75 + 0 = \mathbf{0.75}$
- $\|\mathbf{w}_2\|_1 = 0.25 + 0.5 + 0.25 = 1$

L2 vs. L1 Regularization



L2 regularization will take all information into account to make decisions

L2 vs. L1 Regularization



L1 regularization will focus all the attention to a few key features

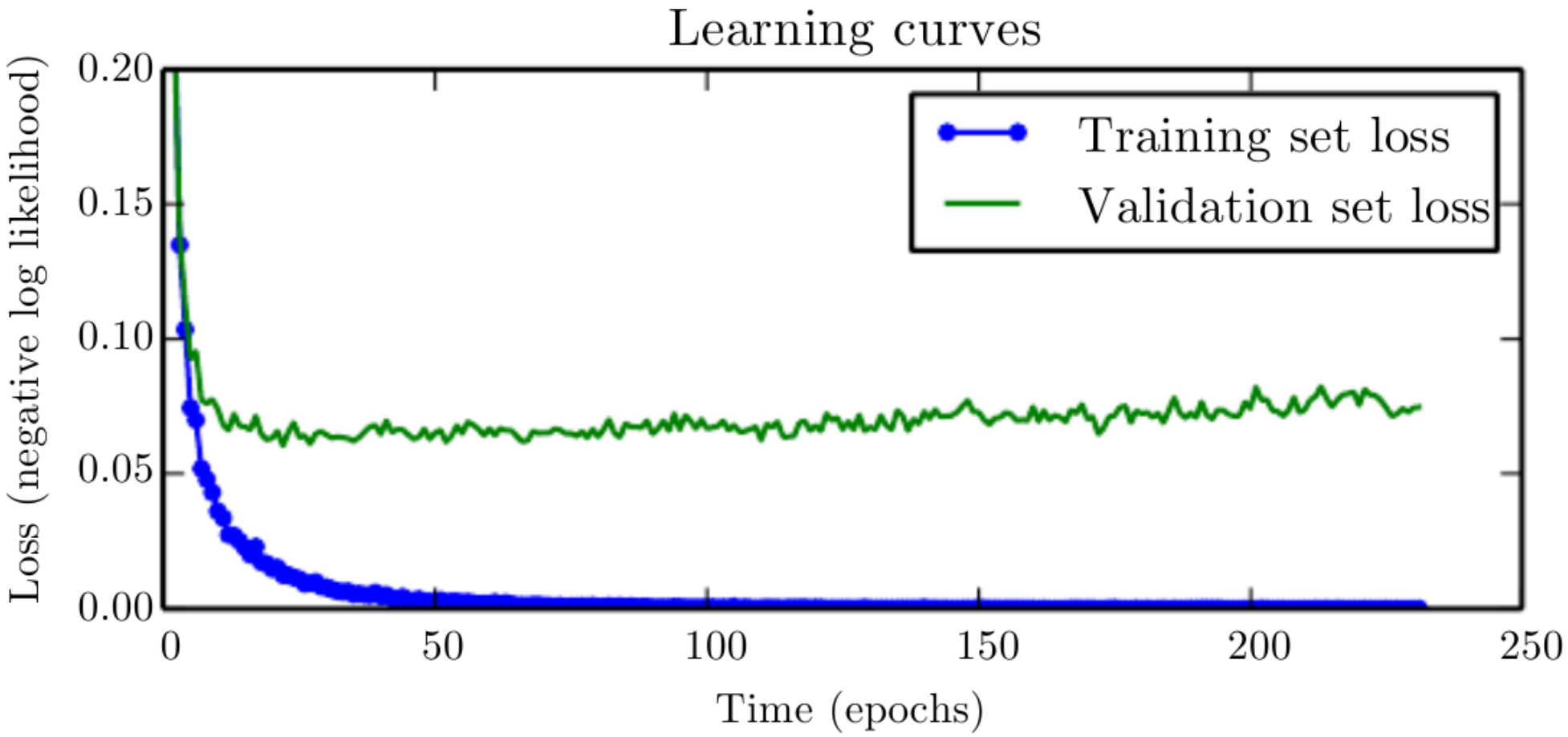
L_2 Regularization for Logistic Regression in PyTorch

Automatically:

```
##################################################
## Apply L2 regularization
optimizer = torch.optim.SGD(model.parameters(),
                            lr=0.1,
                            weight_decay=LAMBDA)
#--------------------------------------------------


for epoch in range(num_epochs):

    #### Compute outputs ####
    out = model(X_train_tensor)

    #### Compute gradients ####
    cost = F.binary_cross_entropy(out, y_train_tensor)
    optimizer.zero_grad()
    cost.backward()
```

Early Stopping

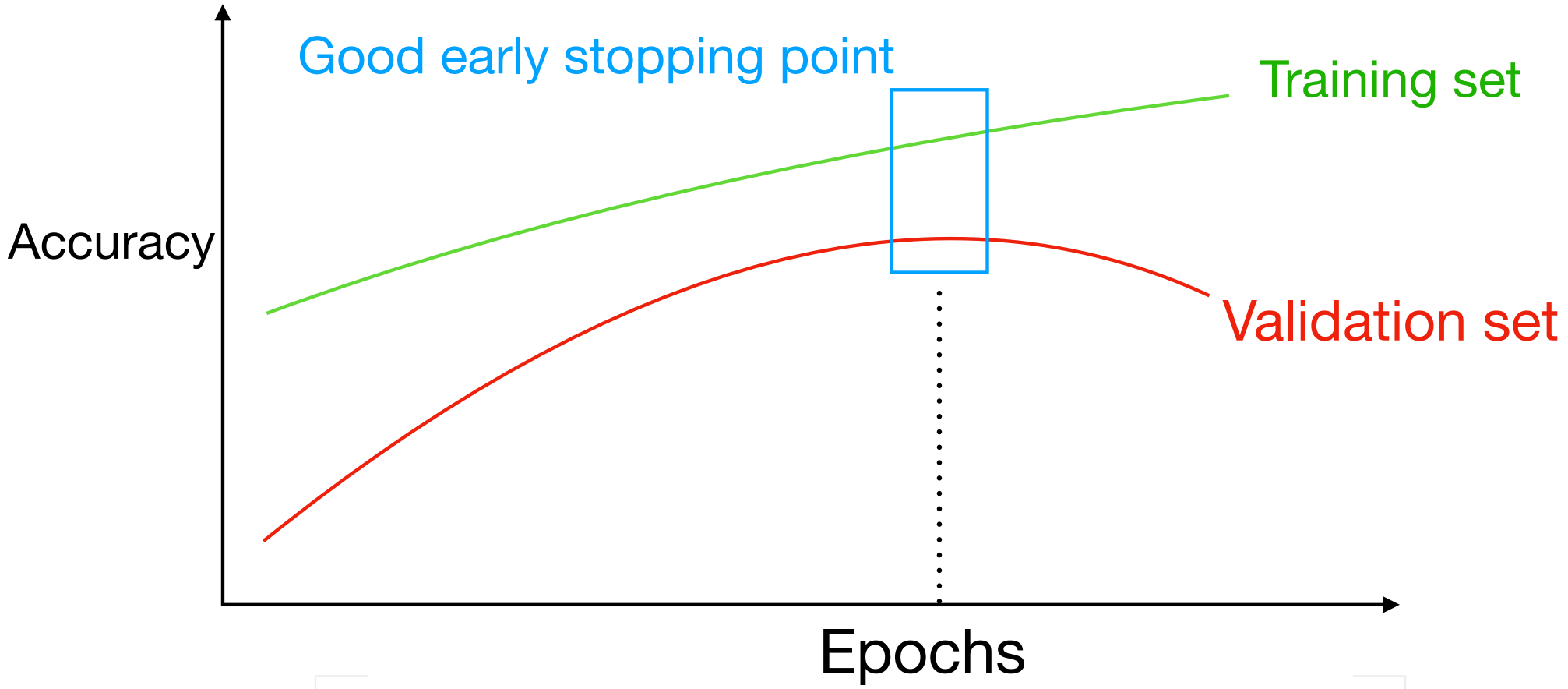
Early Stopping



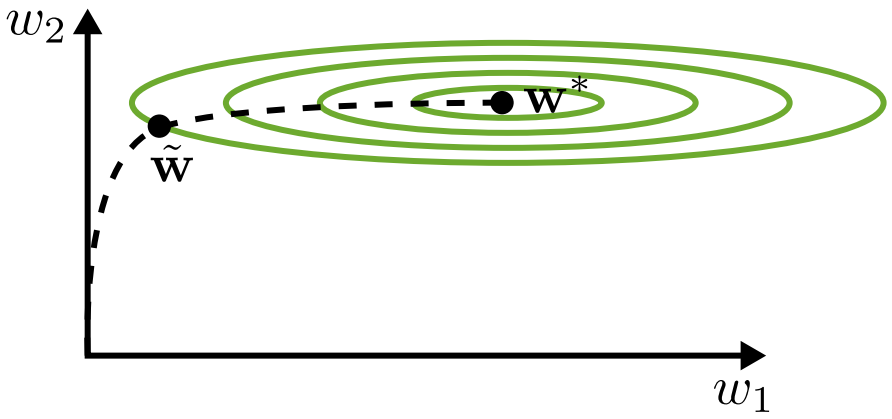
- While training error decreases over time, validation error starts increasing again
- Thus: train for some time and **return parameters with lowest validation error**

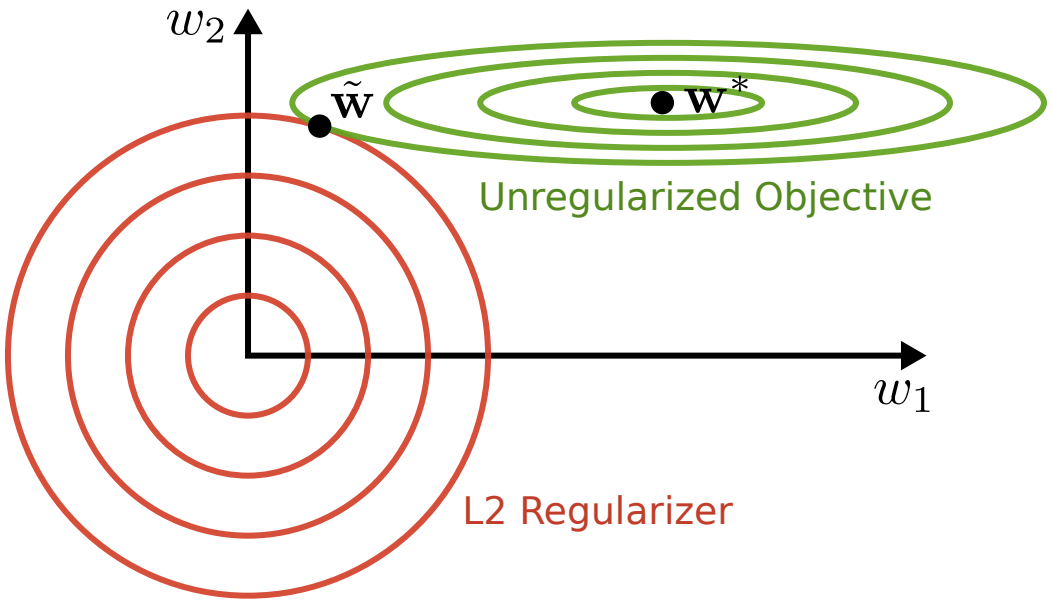
Early Stopping

- reduce overfitting by observing the training/validation accuracy gap during training and then stop at the "right" point



Early Stopping vs. Parameter Penalties





Early stopping:

- Dashed: Trajectory taken by SGD
- Trajectory stops at $\tilde{\mathbf{w}}$ before reaching the minimum $\mathbf{w}^*$

L2 Regularization:

- Regularize objective with L_2 penalty
- Penalty forces minimum of regularized loss $\tilde{\mathbf{w}}$ closer to origin

- Under some assumptions, both are equivalent (see Chapter 7.8 of text book)

Early Stopping

Early Stopping:

- Most commonly used form of regularization in deep learning
- Effective, simple and computationally efficient form of regularization
- Training time can be viewed as hyperparameter $\Rightarrow$ model selection problem
- Efficient as a single training run tests all hyperparameters
- Only cost: periodically evaluate validation error on validation set
- Validation set can be small, and evaluation less frequently

Ensemble Method

Ensemble Methods

Idea:

- Train several models separately for the same task
- At inference time: average results
- Thus, often also called “model averaging”

Intuition:

- Different models make different errors on the test set
- By averaging we obtain a more robust estimate without a better model!

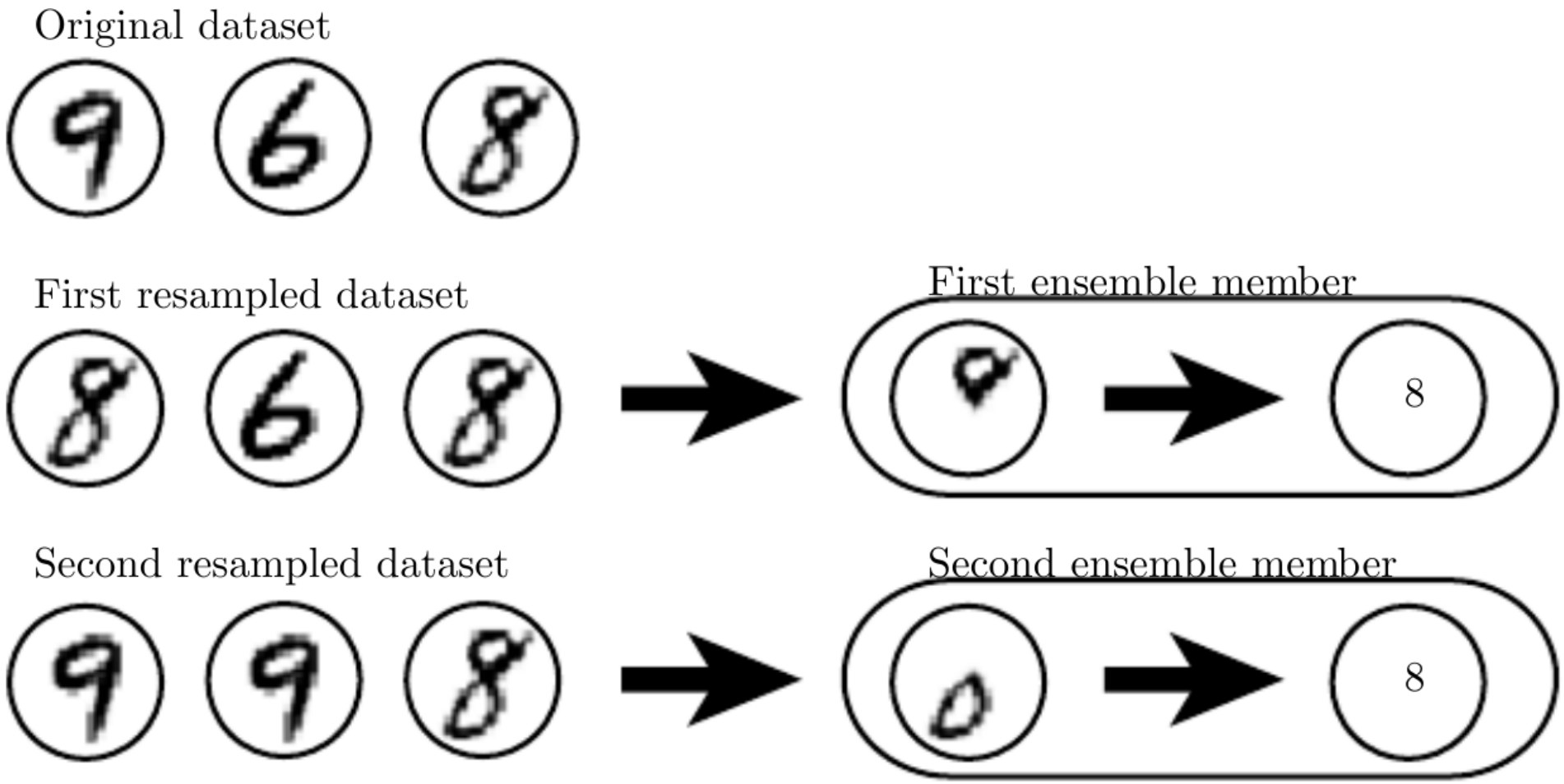
- Drawback: requires evaluation of multiple models at inference time

Ensemble Methods

Different Types of Ensemble Methods:

- **Initialization:** Train networks starting from different random initialization on same dataset or using different minibatches (via stochastic gradient descent). This often already introduces some independence.
- **Model:** Use different models, architectures, losses or hyperparameters
- **Bagging:** Train networks on different random draws (with replacement) from the original dataset. Thus, each dataset likely misses some of the examples from the original dataset and contains some duplicates.

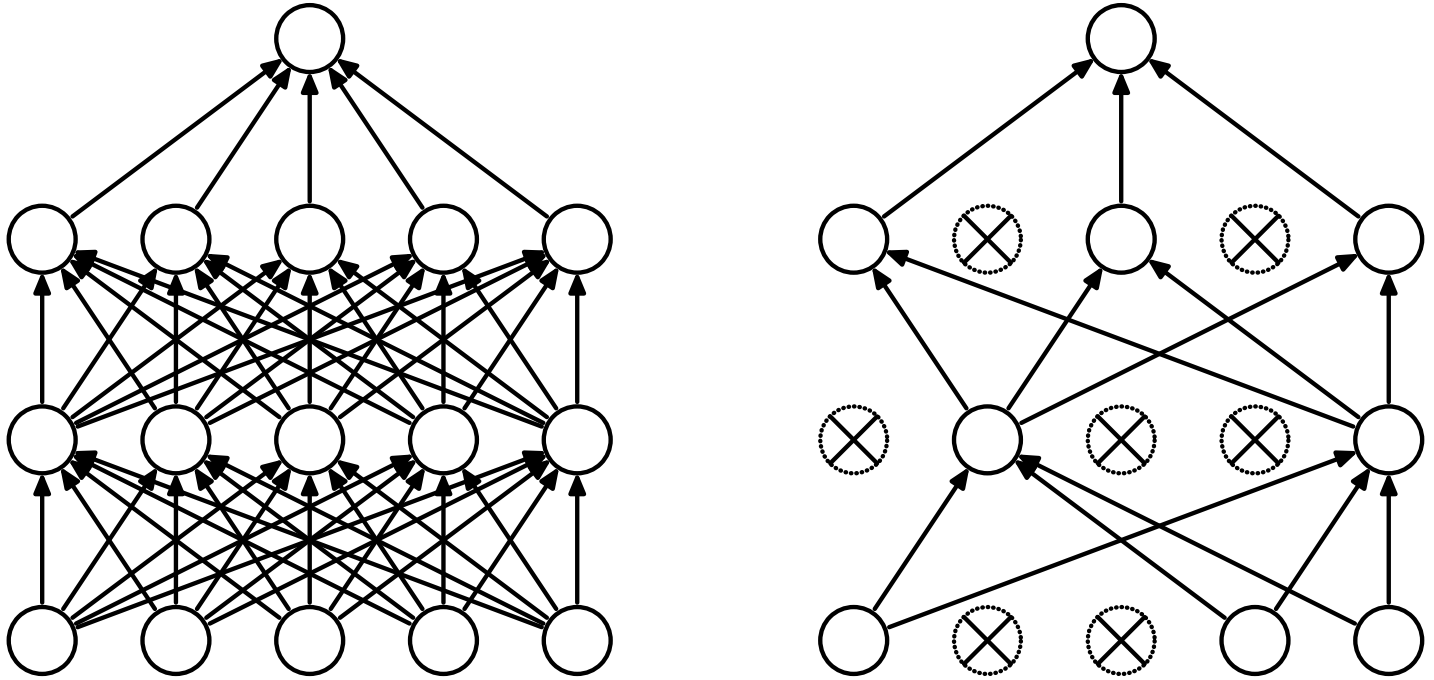
Bagging Example



- First model learns to detect top “loop”, second model detects bottom “loop”

Dropout

Dropout



Idea:

- During training, **set neurons to zero** with probability μ (typically $\mu = 0.5$)
- Each binary mask is one model, changes randomly with every training iteration
- Creates **ensemble "on the fly"** from a single network with shared parameters

Srivastava, Hinton, Krizhevsky, Sutskever and Salakhutdinov: Dropout: a simple way to prevent neural networks from overfitting. JMLR, 2014.

Dropout

How do we drop the nodes practically/efficiently?

Bernoulli Sampling (during training):

- p := drop probability
- $\mathbf{v}$:= random sample from uniform distribution in range [0, 1]
- $\forall i \in \mathbf{v} : v_i := 0 \text{ if } v_i < p \text{ else } 1$
- $\mathbf{a} := \mathbf{a} \odot \mathbf{v}$ *($p \times 100\%$ of the activations a will be zeroed)*

Dropout

How do we drop the nodes practically/efficiently?

Bernoulli Sampling (during training):

- p := drop probability
- $\mathbf{v}$:= random sample from uniform distribution in range [0, 1]
- $\forall i \in \mathbf{v} : v_i := 0 \text{ if } v_i < p \text{ else } 1$
- $\mathbf{a} := \mathbf{a} \odot \mathbf{v}$ *($p \times 100\%$ of the activations a will be zeroed)*

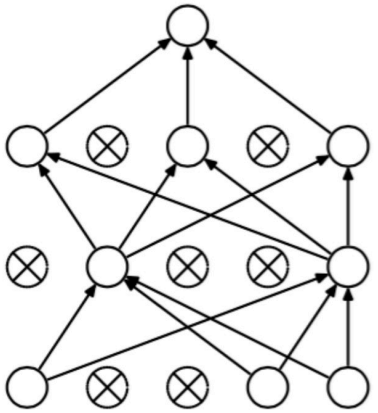
Then, after training when making predictions (during "inference")

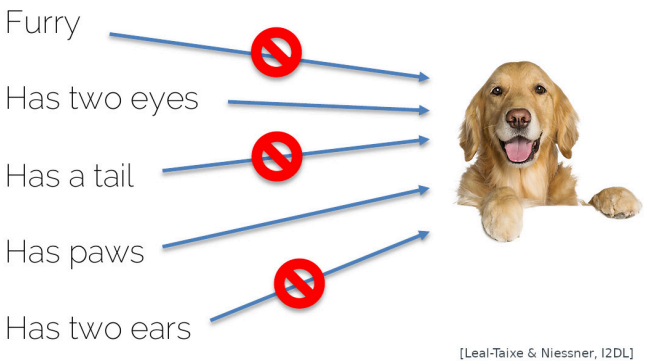
scale activations via $\mathbf{a} := \mathbf{a} \odot (1 - p)$

Q for you: Why is this required?

Dropout: Co-Adaptation Interpretation

Why does Dropout work well?





- Network will learn not to rely on particular connections too heavily
- Thus, will consider more connections (because it cannot rely on individual ones)
- The weight values will be more spread-out (may lead to smaller weights like with L2 norm)
- Side note: You can certainly use different dropout probabilities in different layers (assigning them proportional to the number of units in a layer is not a bad idea, for example)

Dropout: Ensemble Method Interpretation

- In dropout, we have a "different model" for each minibatch
- Via the minibatch iterations, we essentially sample over $M=2^h$ models, where h is the number of hidden units
- Restriction is that we have weight sharing over these models, which can be seen as a form of regularization
- During "inference" we can then average over all these models (but this is very expensive)

Dropout: Ensemble Method Interpretation

- During "inference" we can then average over all these models (but this is very expensive)

This is basically just averaging log likelihoods (this is for one particular class):

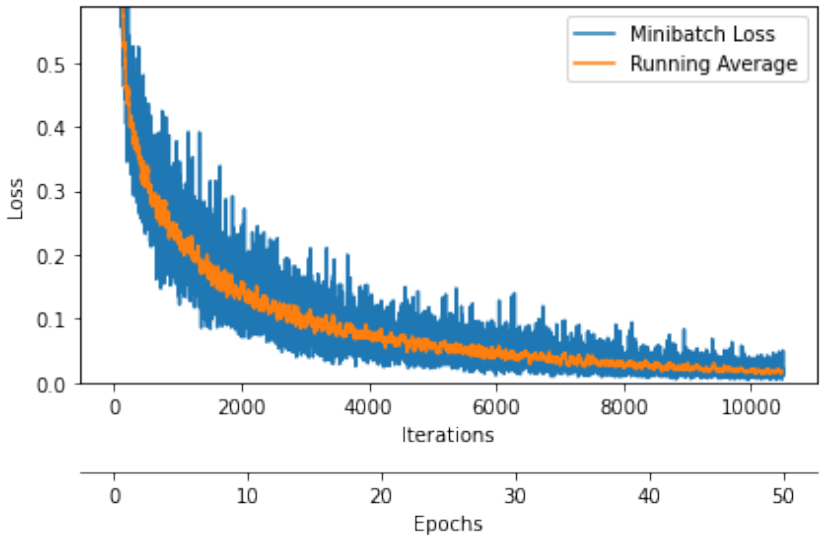
$$p_{\text{Ensemble}} = \Big[\prod_{j=1}^{M} p^{\{i\}}\Big]^{1/M} = \exp\Big[1/M \sum_{j=1}^{M} \log(p^{\{i\}})\Big]$$

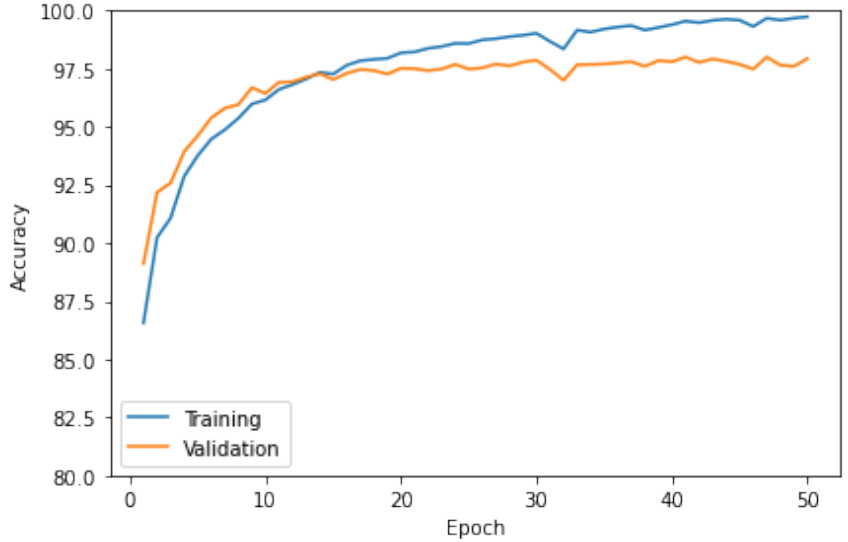
However, using the last model after training and scaling the predictions by a factor *1-p* approximates and is much cheaper

For multiple classes, we need to normalize so that the probas sum to 1:

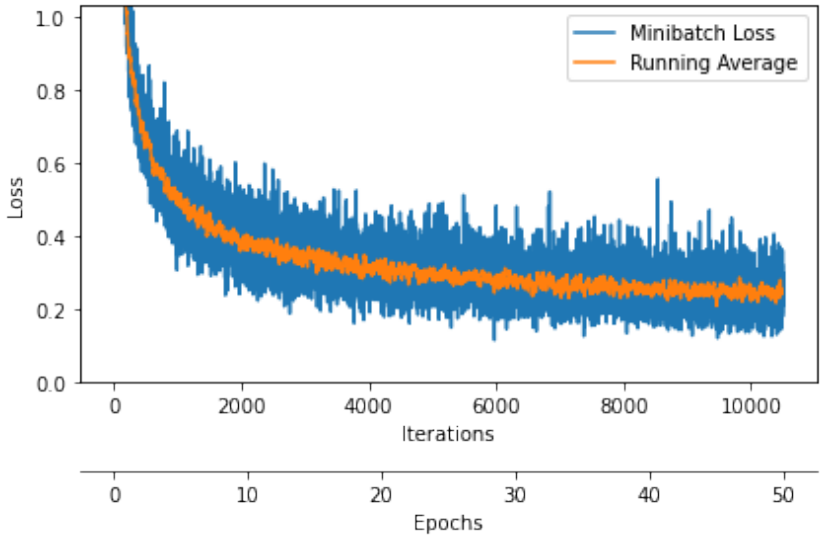
$$p_{\text{Ensemble, j}} = \frac{p_{\text{Ensemble, j}}}{\sum_{j=1}^{k} p_{\text{Ensemble, j}}}$$

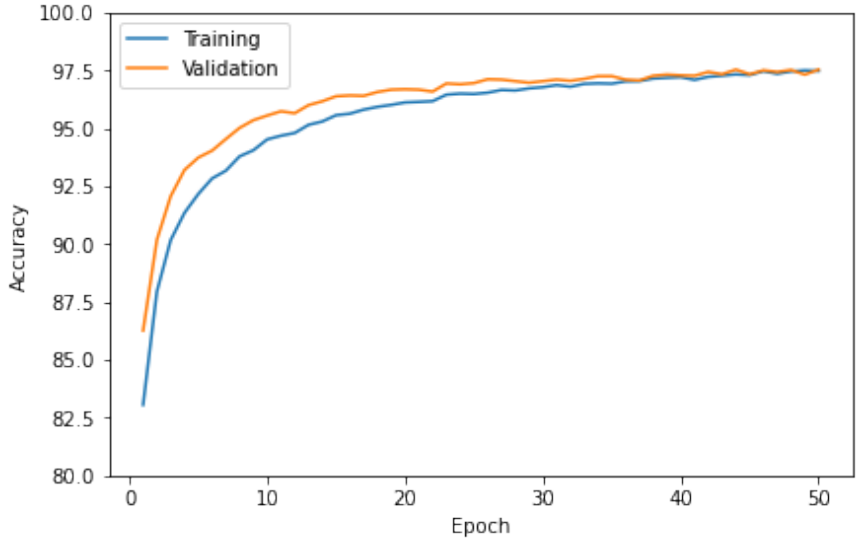
Without dropout:





With 50% dropout:





Dropout

Method	**Test Classification error %**
L2	1.62
L2 + L1 applied towards the end of training	1.60
L2 + KL-sparsity	1.55
Max-norm	1.35
Dropout + L2	1.25
Dropout + Max-norm	**1.05**

Comparison of different regularization methods on MNIST.

Srivastava, Hinton, Krizhevsky, Sutskever and Salakhutdinov: Dropout: a simple way to prevent neural networks from overfitting. JMLR, 2014.

Inverted Dropout

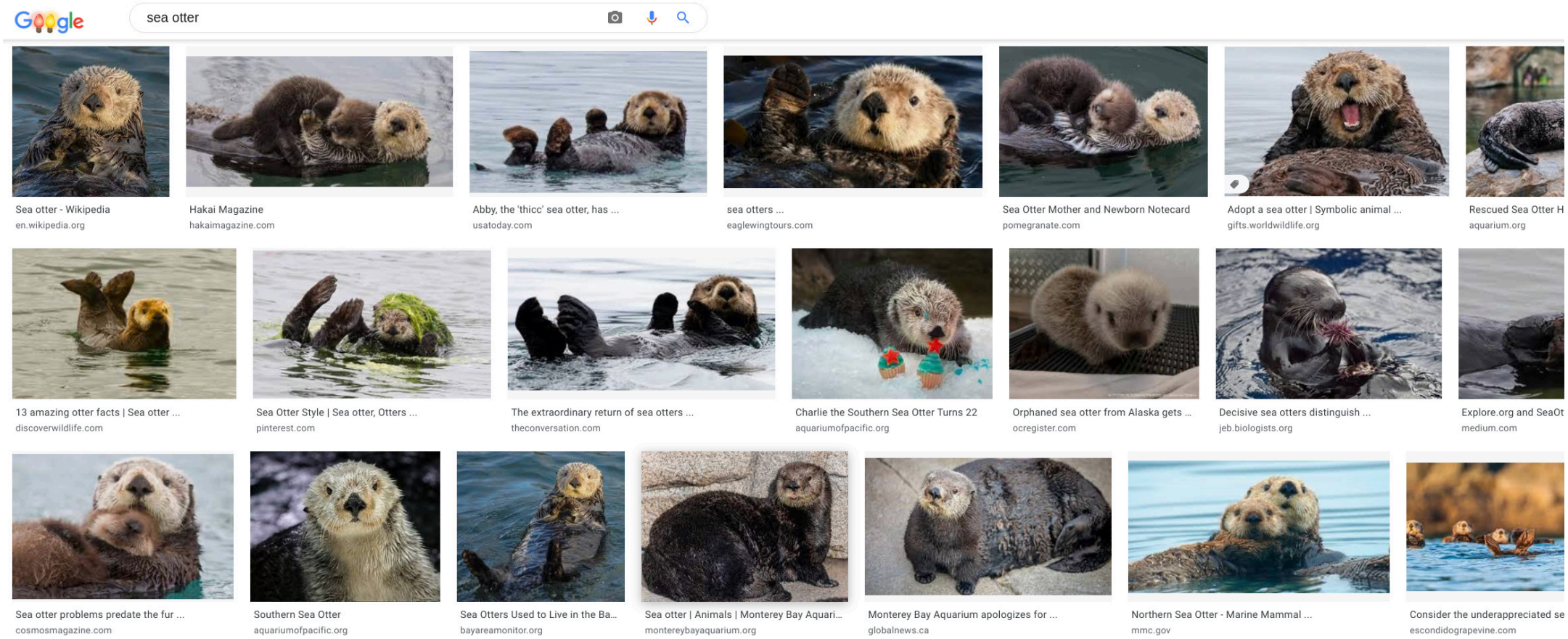
- Most frameworks implement inverted dropout
- Here, the activation values are scaled by the factor *(1-p)* during training instead of scaling the activations during "inference"
- PyTorch's Dropout implementation is also inverted Dropout
- Don't use Dropout if your model does not overfit

Dropout in PyTorch

```
class MultilayerPerceptron(torch.nn.Module):

    def __init__(self, num_features, num_classes, drop_proba,
                 num_hidden_1, num_hidden_2):
        super().__init__()

        self.my_network = torch.nn.Sequential(
            # 1st hidden layer
            torch.nn.Flatten(),
            torch.nn.Linear(num_features, num_hidden_1),
            torch.nn.ReLU(),
            torch.nn.Dropout(drop_proba),
            # 2nd hidden layer
            torch.nn.Linear(num_hidden_1, num_hidden_2),
            torch.nn.ReLU(),
            torch.nn.Dropout(drop_proba),
            # output layer
            torch.nn.Linear(num_hidden_2, num_classes)
        )

    def forward(self, x):
        logits = self.my_network(x)
        return logits
```

Data Augmentation

Data Augmentation

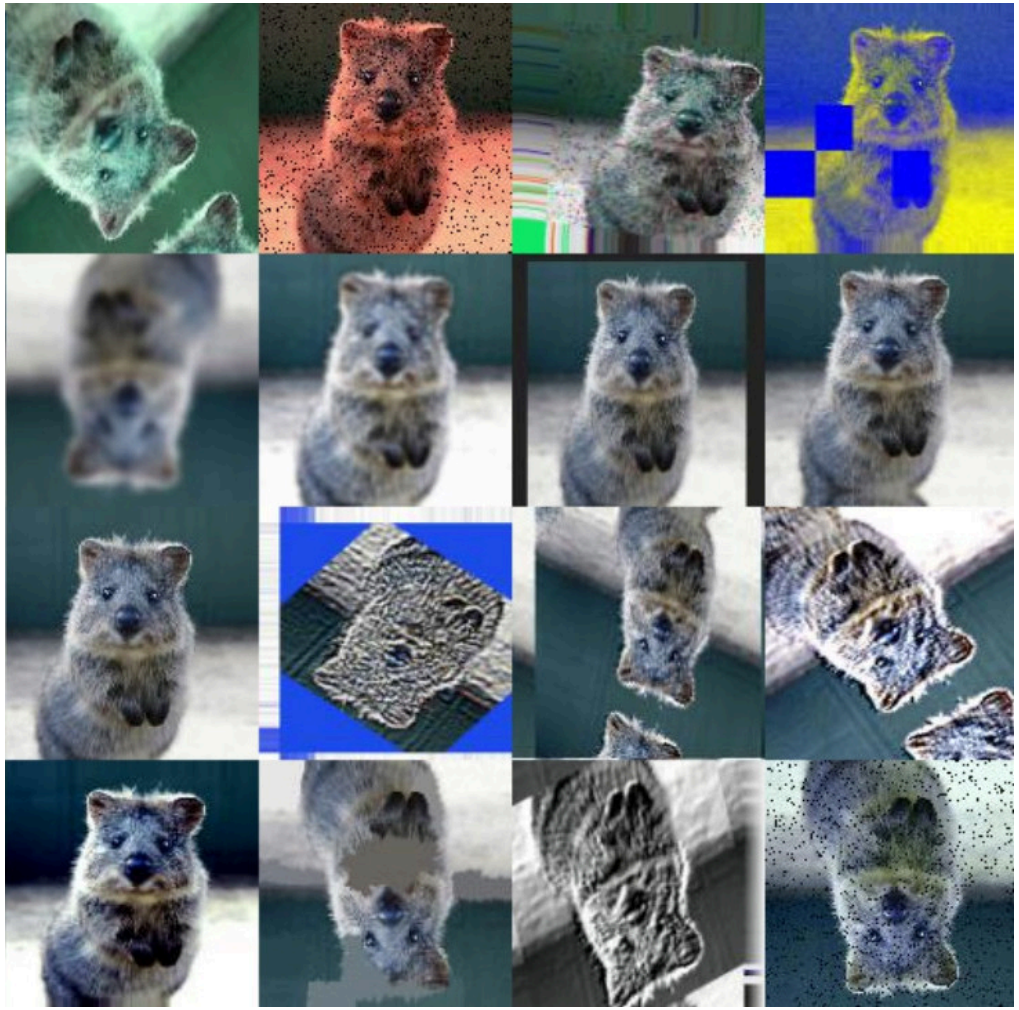


Motivation:

- Deep neural networks must be **invariant** to a wide variety of input variations
- Often **large intra-class variation** in terms of pose, appearance, lighting, etc.

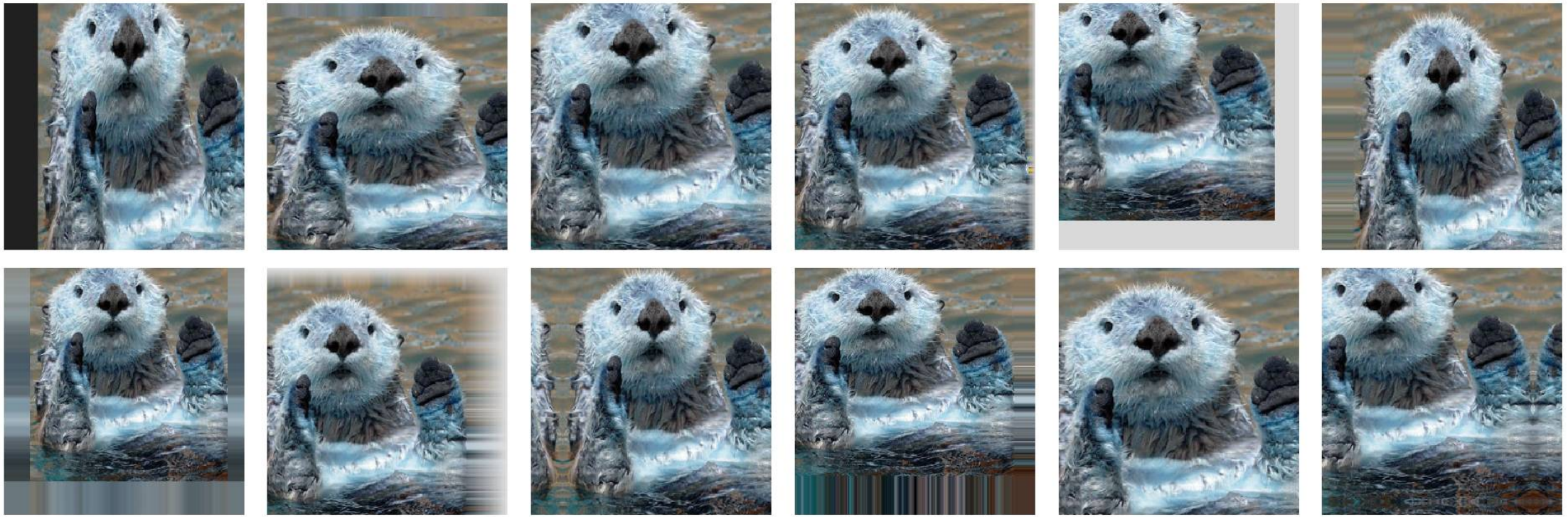
Data Augmentation

- Best way towards better generalization is to **train on more data**
- However, data in practice often limited
- Goal of data augmentation: create **“fake” data** from the existing data (on the fly) and add it to the training set
- New data must **preserve semantics**
- Even **simple operations** like translation or adding per-pixel noise often already greatly improve generalization
- `https://github.com/aleju/imgaug`



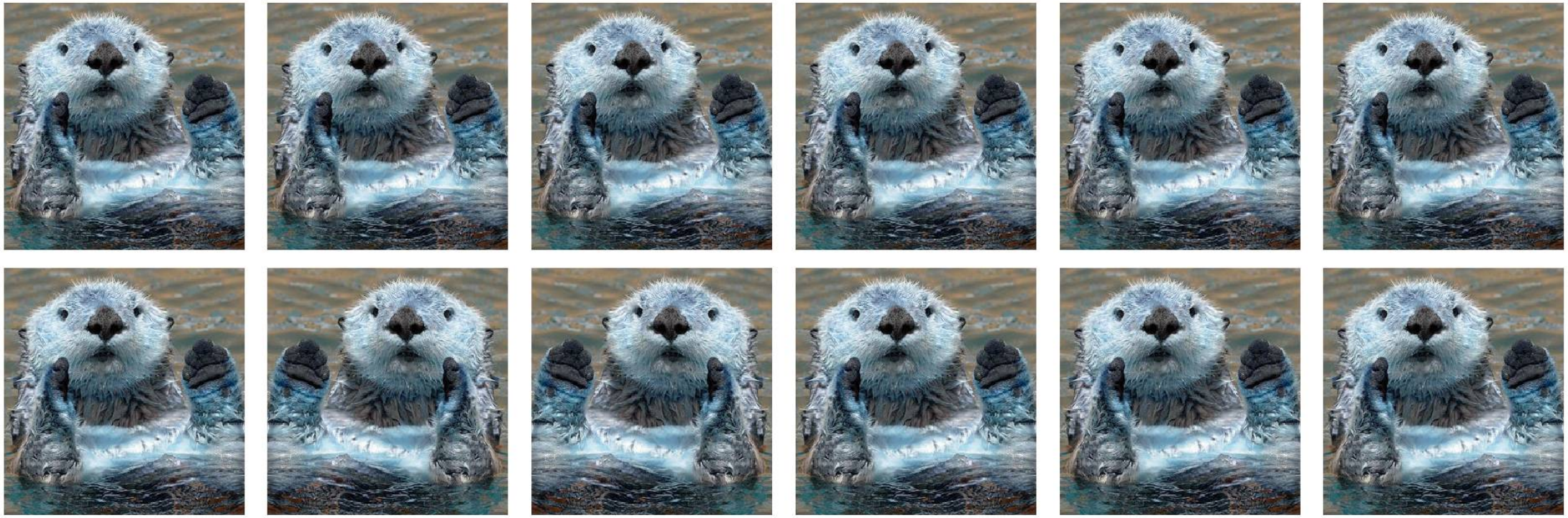
Data Augmentation: Geometry

Image Cropping and Padding



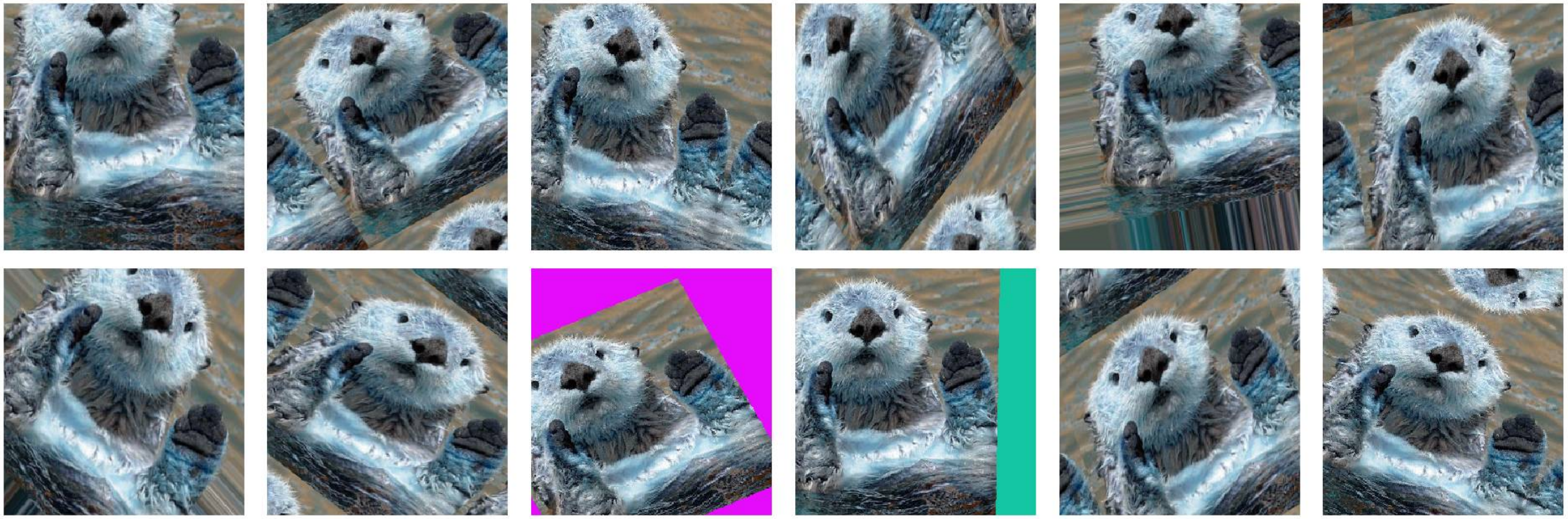
Data Augmentation: Geometry

Horizontal Image Flipping



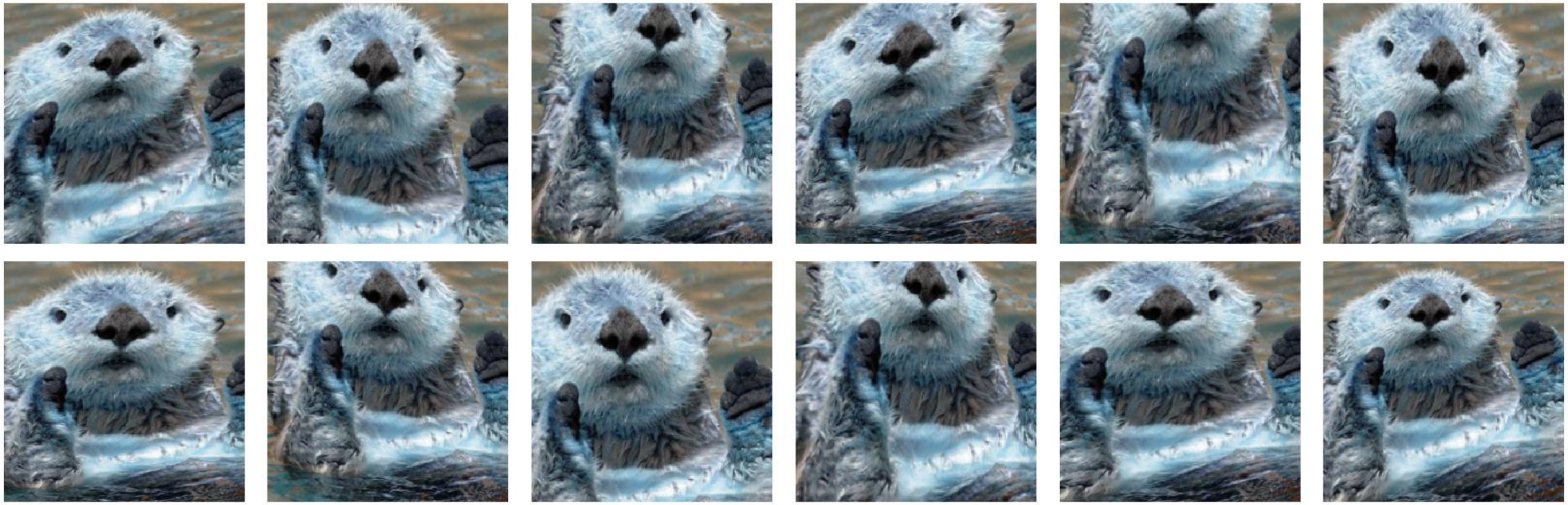
Data Augmentation: Geometry

Affine Transformation



Data Augmentation: Geometry

Perspective Transformation

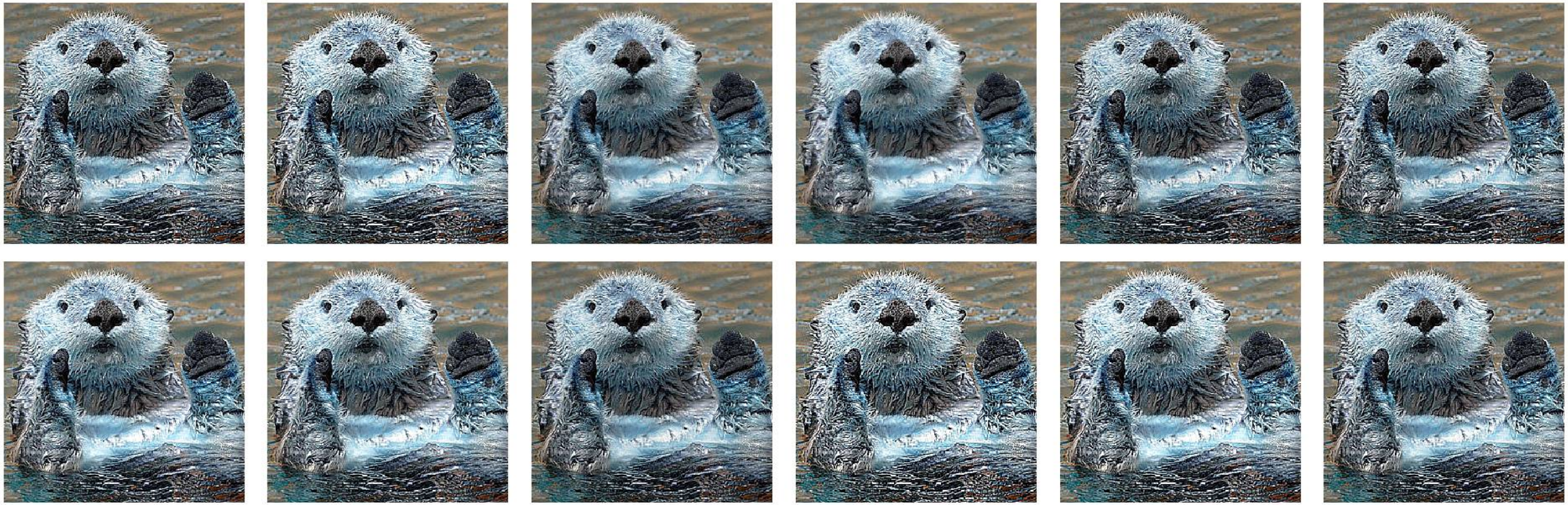


Data Augmentation: Local Filters

Gaussian Blur

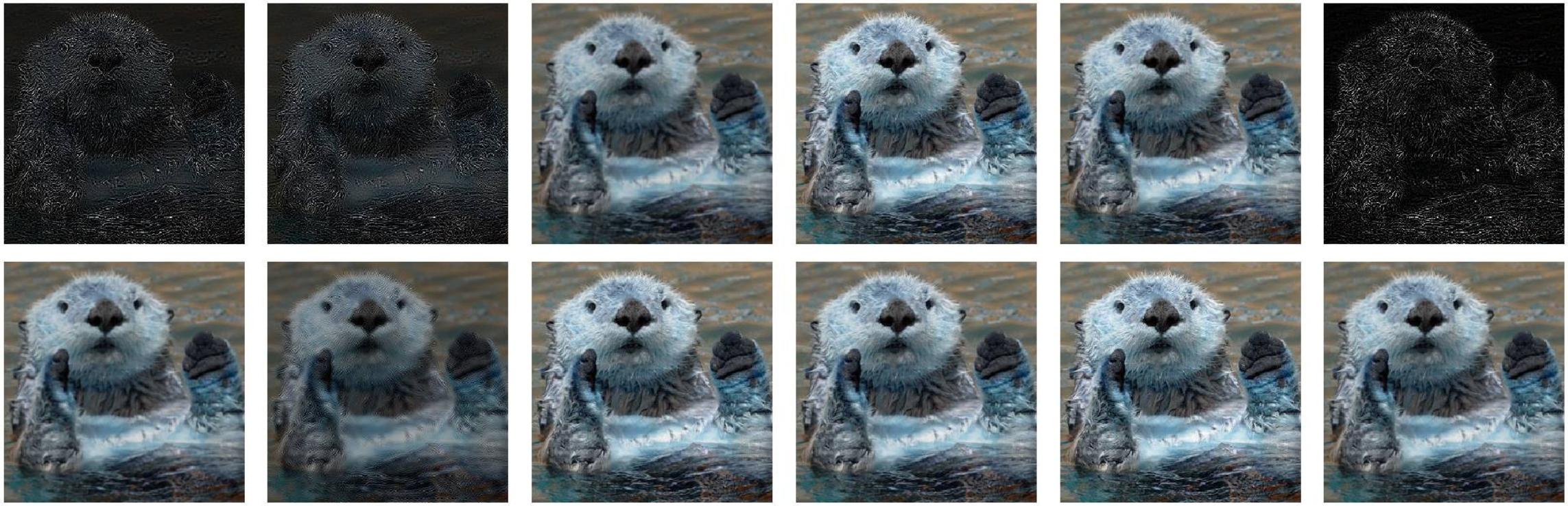
Data Augmentation: Local Filters

Image Sharpening

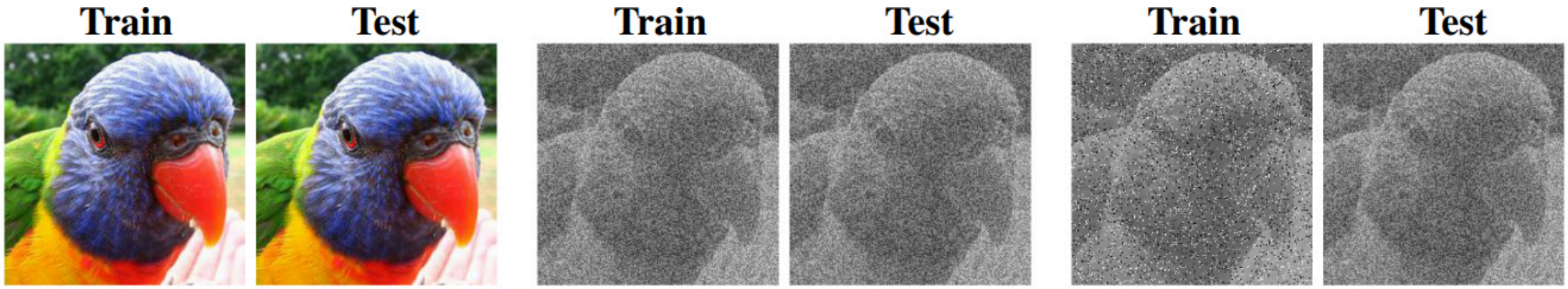


Data Augmentation: Local Filters

Edge Detection



Data Augmentation: Noise



(a) Super-human performance (b) Super-human performance (c) Chance level performance

Figure 1: Classification performance of ResNet-50 trained from scratch on (potentially distorted) ImageNet images. **(a)** Classification performance when trained on standard colour images and tested on colour images is close to perfect (better than human observers). **(b)** Likewise, when trained and tested on images with additive uniform noise, performance is super-human. **(c)** Striking generalisation failure: When trained on images with salt-and-pepper noise and tested on images with uniform noise, performance is at chance level—even though both noise types do not seem much different to human observers.

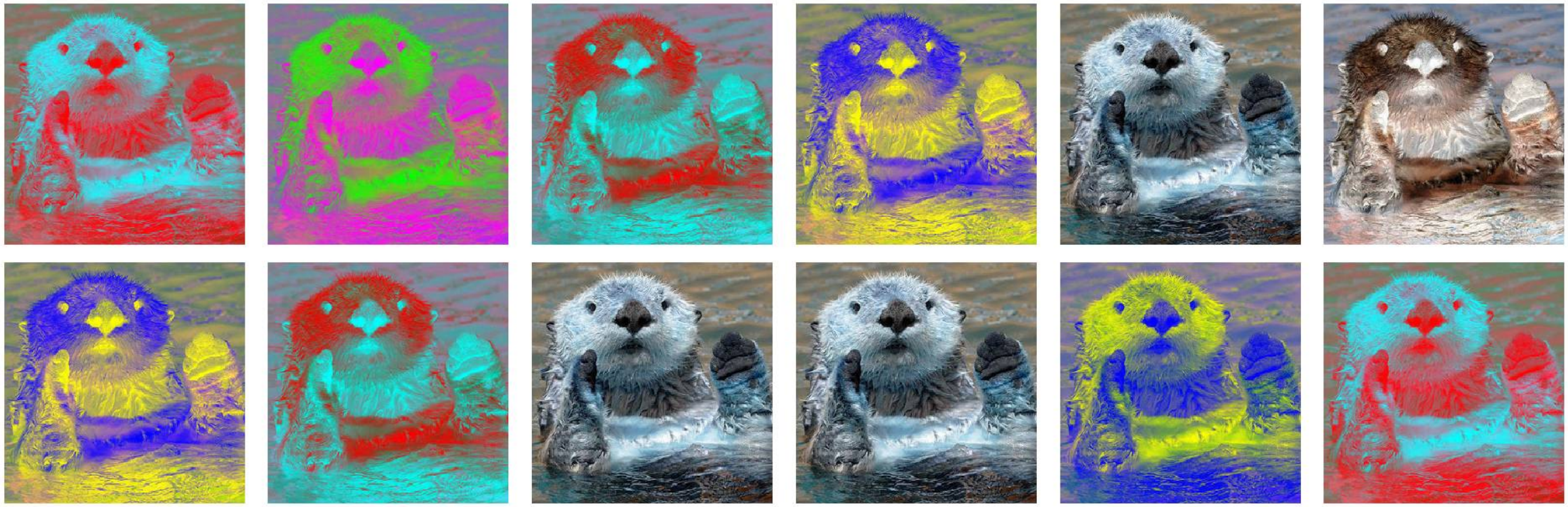
Geirhos, Temme, Rauber, Schütt, Bethge and Wichmann: Generalisation in humans and deep neural networks. NeurIPS, 2018.

Data Augmentation: Color

Hue and Saturation

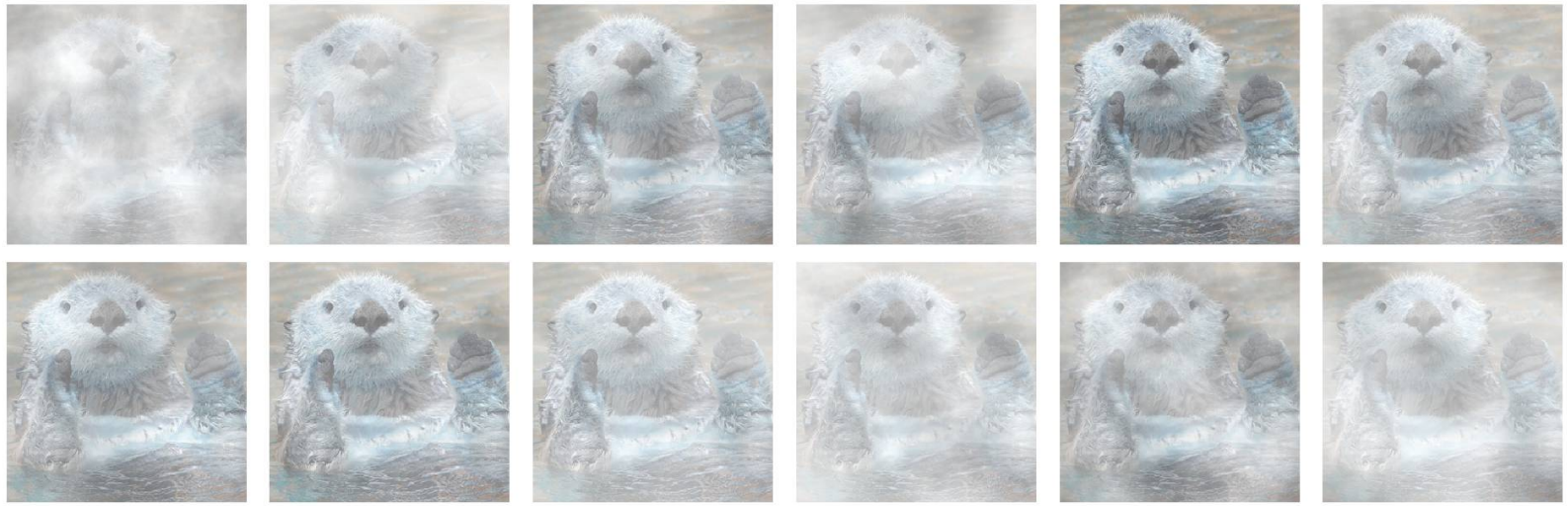
Data Augmentation: Color

Color Inversion

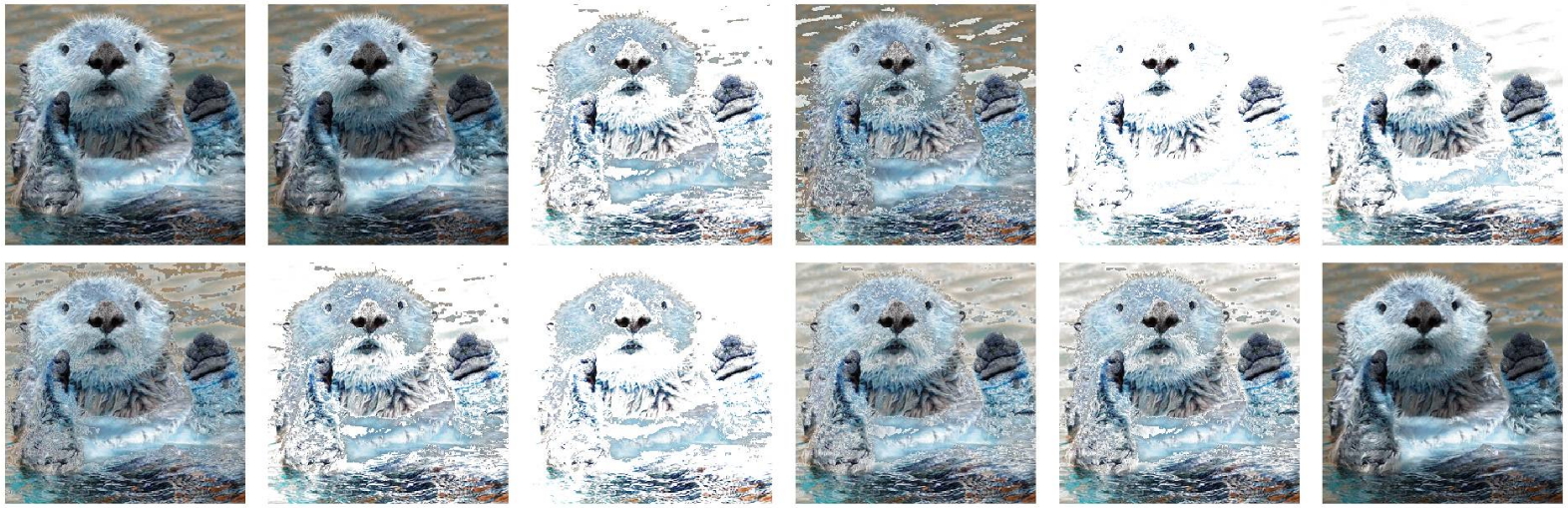


Data Augmentation: Weather

Fog

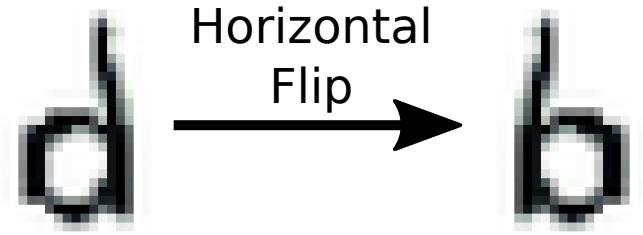


Snow

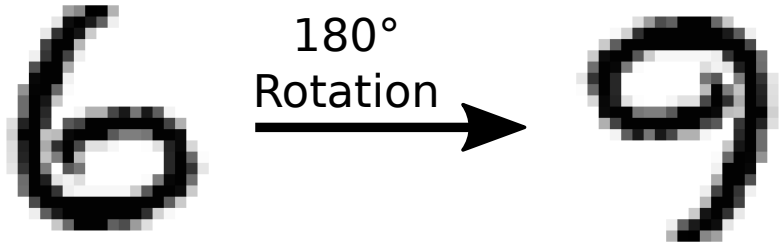


Data Augmentation: Output Transformations

- For some classification tasks, e.g., **handwritten letter recognition**, be careful to not apply transformations that would change the output class
- Example 1: Horizontal flips changes the interpretation of the letter ’d’:



- Example 2: 180° rotations changes the interpretation of the number ’6’:



- Remark: For **general object recognition**, flips and rotations can often be useful!

Next class: Optimizers