Deep Learning Cook Book

Robert Haschke

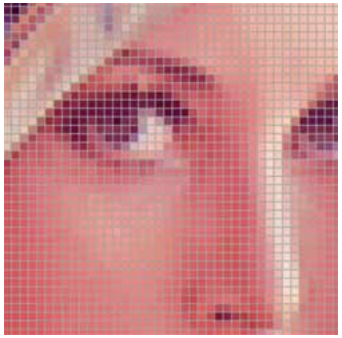
Center of Excellence Cognitive Interaction Technology (CITEC)

Overview

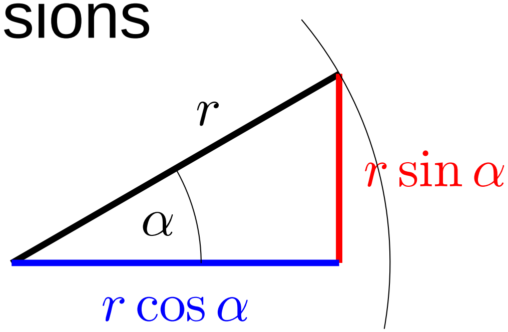
- Input Representation
- Output Layer + Cost Function
- Hidden Layer Units
- Initialization
- Regularization

Input representation

- Choose an input representation that
  - captures / preserves as much structure as possible
    - e.g. images have a grid-like neighbourhood structure
    - preserve that structure instead of flattening the image into a vector

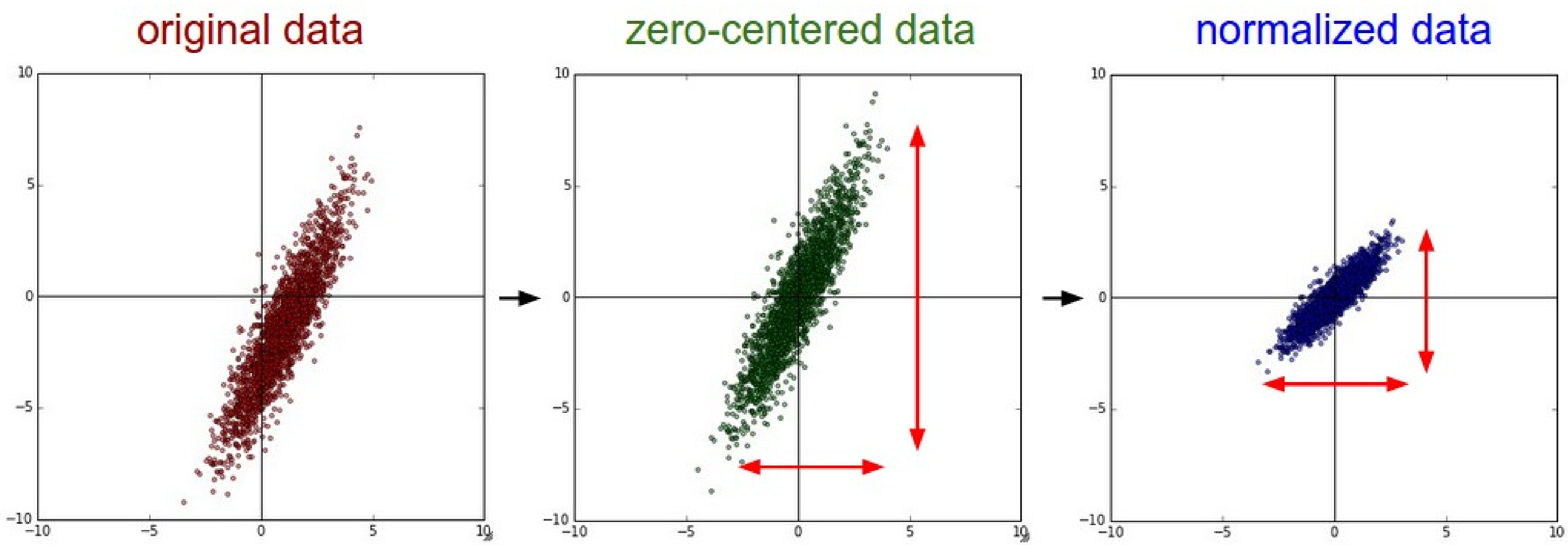


  - changes smoothly with small semantic changes
    - for orientations, use cos/sin expressions instead of angular values



Input Normalization

- Mean subtraction: `X -= np.mean(X, axis=0)`
- Normalization: `X /= np.std(X, axis=0)`

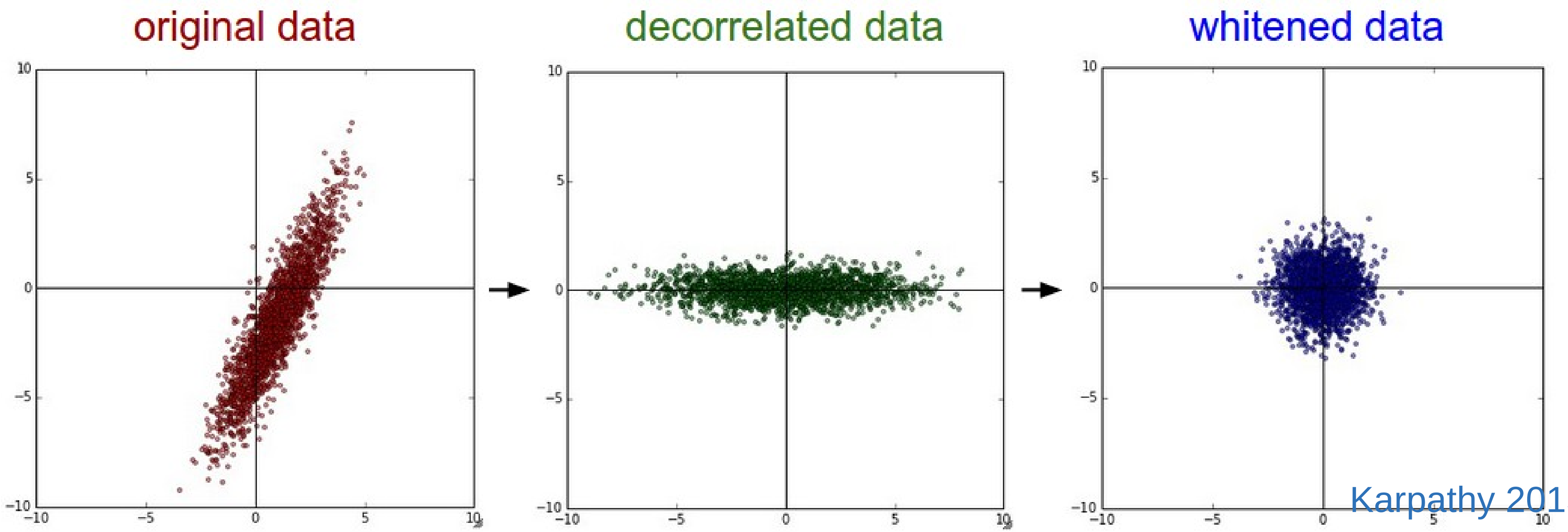


Karpathy 2015

Input Normalization: Whitening / PCA

- Decorrelate feature dimensions with PCA

```
X -= np.mean(X, axis=0) # zero-center
cov = np.dot(X.T, X) / X.shape[0] # covariance
U,S,V = np.linalg.svd(cov)
X_white = np.dot(X, U) / np.sqrt(S)
```



Karpathy 2015

Input Normalization

- Each input feature gets
  - zero mean
  - unit variance
  - Decorrelated
- … across whole *training* data set
- Only *compute* normalization on *training* data set.
- *Apply* calculated transformation to test + validation set.

Output Units and Cost Functions

- How to choose a suitable output layer and cost function for a given task?

- Output layer encodes result
- Cost function defines learning task

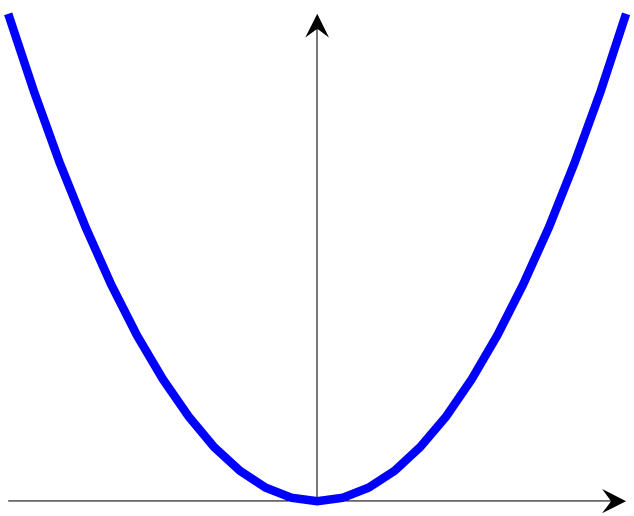
- avoid saturation = vanishing gradient

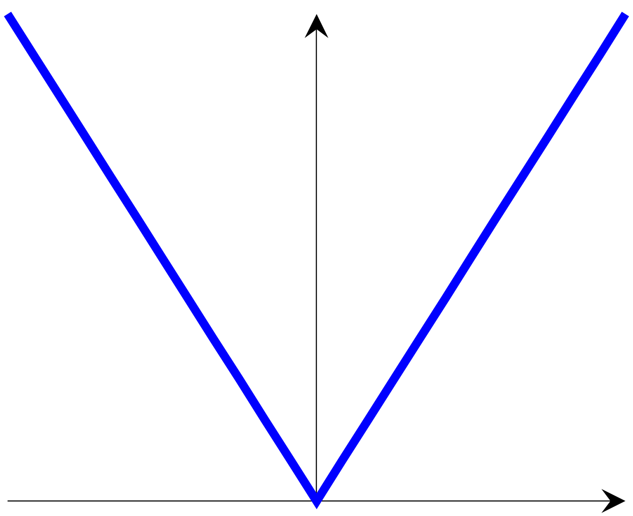
Maximum Likelihood Optimization

- Maximize Likelihood: $\max_{\boldsymbol{\theta}} p(\mathbf{y}|\mathbf{x})$
- Often a Gaussian error model is assumed:
  $p(\mathbf{y}|\mathbf{x}) = \mathcal{N}(\mathbf{y}; \boldsymbol{\mu} = f(\mathbf{x}, \boldsymbol{\theta}), \Sigma = \mathbf{1})$
- This results in minimization of **mean squared error**:
  $J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x},\mathbf{y}\sim p} \|\mathbf{y}^\alpha - f(\mathbf{x}^\alpha, \boldsymbol{\theta})\|^2 + \text{const}$
- If we could train on infinitely many samples and our model is rich enough, we would get:
  $f^*(\mathbf{x}) = \mathbb{E}_{\mathbf{y}\sim p(\mathbf{y}|\mathbf{x})}\, \mathbf{y}$

L_1 vs. L_2 norm

- Using L_1 norm instead of L_2 norm (**mean absolute error**) $J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x},\mathbf{y}\sim p} \|\mathbf{y}^\alpha - f(\mathbf{x}^\alpha, \boldsymbol{\theta})\|_1$
- $f^*(\mathbf{x})$ would resemble the *median* of $\mathbf{y}$ given $\mathbf{x}$





Linear Outputs + MSE / MAE

- Linear outputs: $\hat{\mathbf{y}} = W\mathbf{h} + \mathbf{b}$
- suitable for training *unconstrained* outputs
- train with MSE or MAE for cost
- gradient is linear
  - MSE / L_2-norm: $\nabla_{\mathbf{h}} J = W^t(\mathbf{y} - f(x))$
  - MAE / L_1-norm: $\nabla_{\mathbf{h}} J = W^t(\pm 1, \pm 1, \ldots, \pm 1)^t$

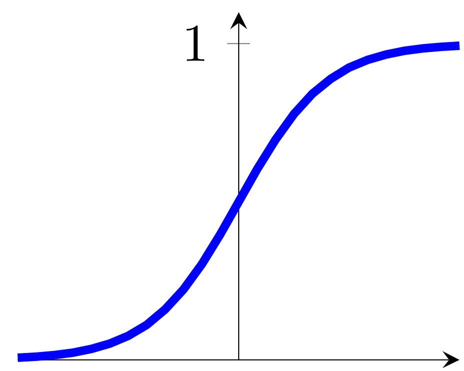
Binary Classification Task

- Task: classify between two classes (0,1)
- Approach: Predict $p(\mathbf{y} \mid \mathbf{x})$ with *Bernoulli distribution*
  - $p(\mathbf{y}=1 \mid \mathbf{x}) = p \approx \sigma(w \cdot x + b)$
  - $p(\mathbf{y}=0 \mid \mathbf{x}) = 1 - p$

Single output suffices!

- sigmoid $\sigma(\cdot)$ ensures constraint $p \in [0,1]$
- $\min -\log \sigma(\cdot)$ reverts „squashing“

$$
\begin{aligned}
-log\,\sigma(x) &= -(log 1 - log(1 + exp(-x))) \\
&= log(1 + exp(-x)) \equiv \zeta(-x)
\end{aligned}
$$



$$\sigma(x) = \frac{1}{1 + exp(-x)}$$

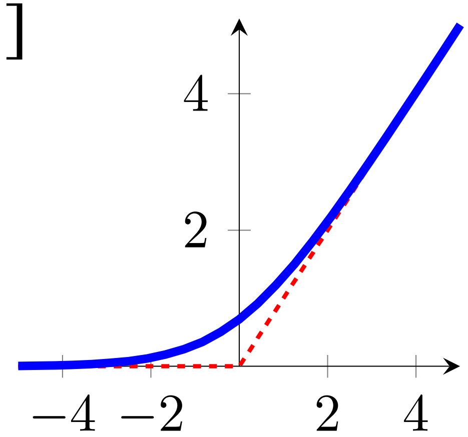
Binary Classification Task

- Task: classify between two classes (0,1)
- Approach: Predict $p(\mathbf{y} \mid \mathbf{x})$ with *Bernoulli distribution*
  - $p(\mathbf{y}=1 \mid \mathbf{x}) = p \approx \sigma(w \cdot x + b)$
  - $p(\mathbf{y}=0 \mid \mathbf{x}) = 1\text{-}p$

Single output suffices!

- sigmoid $\sigma(\cdot)$ ensures constraint $p \in [0,1]$
- $\min -\log \sigma(\cdot)$ reverts „squashing“

$$
\begin{aligned}
-log\,\sigma(x) &= -(log 1 - log(1 + exp(-x)) \\
&= log(1 + exp(-x)) \equiv \zeta(-x)
\end{aligned}
$$



$\zeta(x) = log(1 + exp(x))$

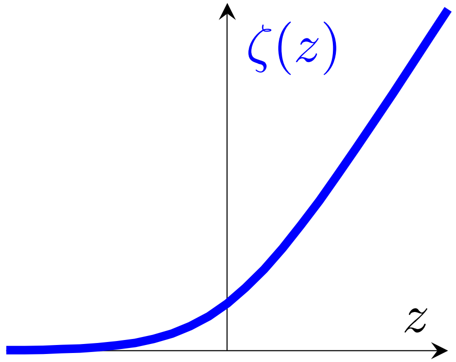
soft-plus function: soft version of $x^{+} = \max(0,x)$

Binary Classification Task

- for binary classification use
  - (single) sigmoid output
  - **binary cross-entropy** as cost

$$J(y,\hat{y}) = -\left(y \cdot log\,\hat{y} + (1-y)\cdot log(1-\hat{y})\right)$$

$$= \begin{cases} -log\,\sigma(z) = \zeta(-z) & \text{if } y = 1 \\ -log(1-\sigma(z)) = \zeta(z) & \text{if } y = 0 \end{cases}$$



  - because target y $\in \{0,1\}$
- Gradient gets small only if classification is correct.

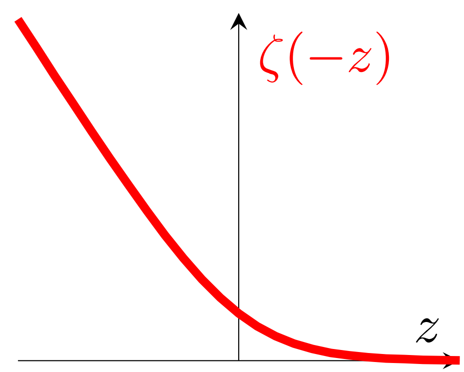
Binary Classification Task

- for binary classification use
  - (single) sigmoid output
  - **binary cross-entropy** as cost

$$J(y,\hat{y}) = -\left(y \cdot log\,\hat{y} + (1-y)\cdot log(1-\hat{y})\right)$$

$$= \begin{cases} -log\,\sigma(z) = \zeta(-z) & \text{if } {\color{red} y = 1} \\ -log(1-\sigma(z)) = \zeta(z) & \text{if } {\color{blue} y = 0} \end{cases}$$

  - because target y $\in \{0,1\}$
- Gradient gets small only if classification is correct.

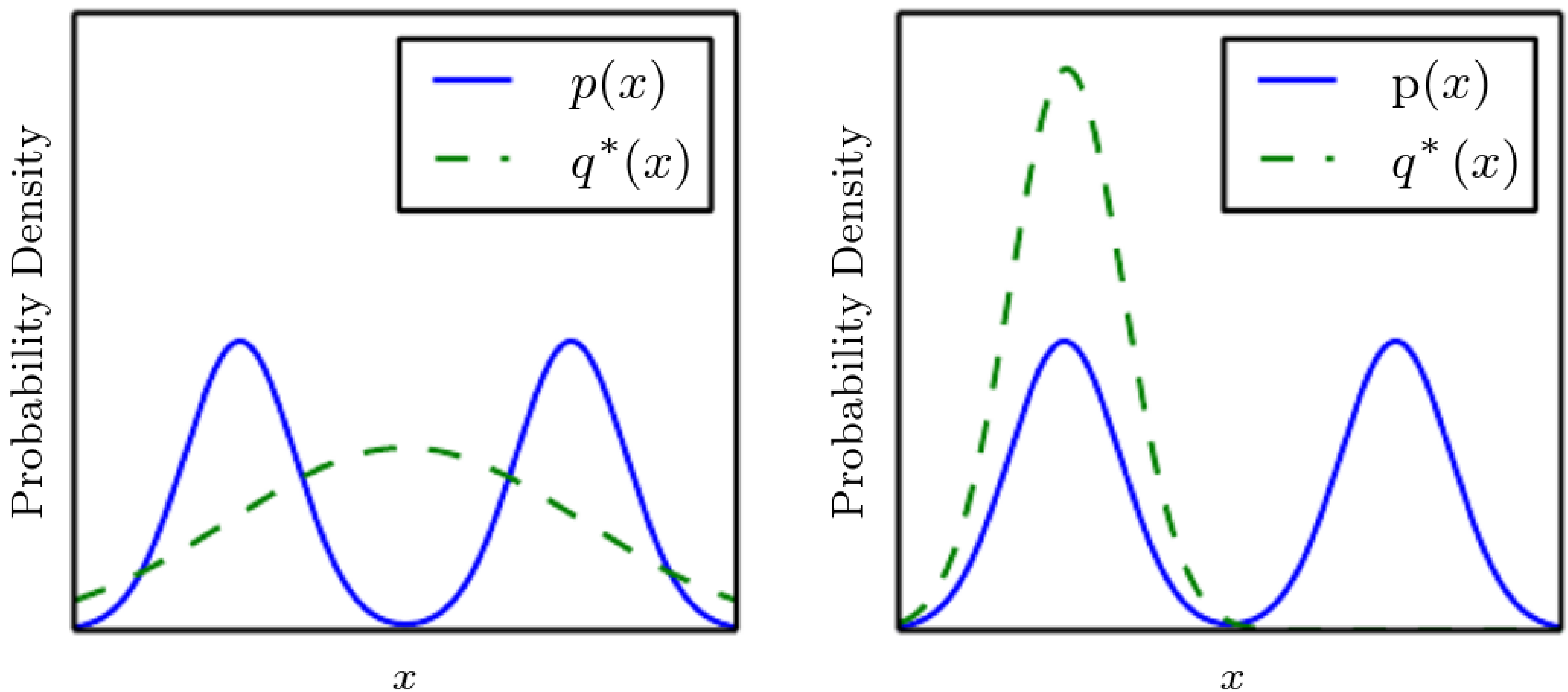


Kullback-Leibler divergence

- $$D_{KL}(P \parallel Q) = \mathbb{E}_{\mathbf{x}\sim P}[log\, P(\mathbf{x}) - log\, Q(\mathbf{x})]$$
  $$= \int p(\mathbf{x})\, log\, \frac{p(\mathbf{x})}{q(\mathbf{x})} d\mathbf{x} \quad \text{continuous distribution}$$
  $$= \sum_{\mathbf{x}} P(\mathbf{x})\, log\, \frac{P(\mathbf{x})}{Q(\mathbf{x})} \quad \text{discrete distribution}$$
- measures *difference* between two distributions P and Q
- $D_{KL}(P \parallel Q) \geq 0$, equality iff $P \equiv Q$
- KL divergence is asymmetric:
  $D_{KL}(P \parallel Q) \neq D_{KL}(Q \parallel P)$

Asymmetry of KL divergence

$$q^* = \text{argmin}_q D_{\text{KL}}(p \| q) \qquad\qquad q^* = \text{argmin}_q D_{\text{KL}}(q \| p)$$



Cross-Entropy

- Minimizing KL-divergence $D_{KL}(P \,||\, Q)$ w.r.t. Q is equivalent to minimizing the **cross-entropy** $H(P \,||\, Q)$:

$$H(P \parallel Q) = -\mathbb{E}_{\mathbf{x}\sim P}\, log\, Q(\mathbf{x}) = H(P) + D_{KL}(P \parallel Q)$$

$$H(P) = -\mathbb{E}_{\mathbf{x}\sim P}\, log\, P(\mathbf{x})$$

$$D_{KL}(P \parallel Q) = \mathbb{E}_{\mathbf{x}\sim P}[log\, P(\mathbf{x}) - log\, Q(\mathbf{x})]$$

- binary cross-entropy (of Bernoulli distributions)

$$H(p \parallel q) = -\left(p \cdot log\, q + (1-p) \cdot log(1-q)\right)$$

From 2-Class to Multi-Class Classification

- Multi-Class classification should output a probability distribution over all N classes:
  $y_i = P(y = i \mid \mathbf{x})$
- with constraints
  - $y_i \in [0,1]$
  - $\Sigma_i\, y_i = 1$
- Soft-Max on $\mathbf{z} = W\,\mathbf{h} + \mathbf{b}$ ensures these constraints:

$$\text{soft-max}(\mathbf{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

- Minimizing cross-entropy again reverts the exponential…

Minimizing Cross-Entropy

- $\log \text{soft-max}(\mathbf{z})_i = z_i - \log \sum_j \exp(z_j)$
- z_i contributes linearly to cost
- $\log \sum_j \exp(z_j) \approx \max_j z_j$
- If classification is already correct, i.e. $\operatorname{argmax}_j z_j = i$, the terms cancel out.
  The example doesn't contribute to the cost.
- Special Case: 2D Soft-Max:
  $$y_1 = \frac{\exp(z_1)}{\exp(z_1) + \exp(z_2)} = \frac{1}{1 + \exp(z_2 - z_1)} = \sigma(z_1 - z_2)$$

Gradient of Cross-Entropy + Soft-Max

$$H(P \parallel \mathbf{y}) = -\sum_{\alpha}\sum_{i} P_i(\mathbf{x}^\alpha) \, \log \, y_i(\mathbf{x}^\alpha)$$

$$y_k = \frac{\exp(z_k)}{\sum_j \exp(z_j)}$$

$$\mathbf{z} = W\,\mathbf{x} + \mathbf{b}$$

$$-\frac{\partial H}{\partial w_{ij}} = -\sum_k \frac{\partial H}{\partial y_k} \cdot \frac{\partial y_k}{\partial z_i} \cdot \frac{\partial z_i}{\partial w_{ij}}$$

Gradient of Cross-Entropy + Soft-Max

$$H(P \parallel \mathbf{y}) = -\sum_{\alpha}\sum_{i} P_i(\mathbf{x}^{\alpha}) \log y_i(\mathbf{x}^{\alpha})$$

$$\frac{\partial H}{\partial y_k} = -\sum_{\alpha} P_k(x^{\alpha}) \cdot \frac{1}{y_k^{\alpha}}$$

$$\frac{\partial H}{\partial y_k} = -\sum_{\alpha} \frac{P_k^{\alpha}}{y_k^{\alpha}}$$

Gradient of Cross-Entropy + Soft-Max

$$H(P \parallel \mathbf{y}) = -\sum_{\alpha} \sum_{i} P_i(\mathbf{x}^{\alpha}) \, \log \, y_i(\mathbf{x}^{\alpha})$$

$$y_k = \frac{\exp(z_k)}{\sum_j \exp(z_j)}$$

$$\frac{\partial y_k}{\partial z_i} = \frac{\delta_{ki} e^{z_k} \cdot N - e^{z_i} \cdot e^{z_k}}{N^2}$$

$$= \delta_{ki} \frac{e^{z_k}}{N} - \frac{e^{z_i}}{N} \cdot \frac{e^{z_k}}{N}$$

$$= \delta_{ki} y_k - y_i \cdot y_k$$

$$\frac{\partial H}{\partial y_k} = -\sum_{\alpha} \frac{P_k^{\alpha}}{y_k^{\alpha}}$$

$$\frac{\partial y_k}{\partial z_i} = y_k(\delta_{ki} - y_i)$$

Gradient of Cross-Entropy + Soft-Max

$$H(P \parallel \mathbf{y}) = -\sum_{\alpha}\sum_{i} P_i(\mathbf{x}^\alpha)\,\log\, y_i(\mathbf{x}^\alpha)$$

$$y_k = \frac{\exp(z_k)}{\sum_j \exp(z_j)}$$

$$\mathbf{z} = W\,\mathbf{x} + \mathbf{b}$$

$$\frac{\partial z_i}{\partial w_{ij}} = x_j$$

$$\frac{\partial H}{\partial y_k} = -\sum_{\alpha} \frac{P_k^\alpha}{y_k^\alpha}$$

$$\frac{\partial y_k}{\partial z_i} = y_k(\delta_{ki} - y_i)$$

$$\frac{\partial z_i}{\partial w_{ij}} = x_j$$

Gradient of Cross-Entropy + Soft-Max

$$
\begin{aligned}
-\frac{\partial H}{\partial w_{ij}} &= \sum_k -\frac{\partial H}{\partial y_k} \cdot \frac{\partial y_k}{\partial z_i} \cdot \frac{\partial z_i}{\partial w_{ij}} \\
&= \sum_k \sum_\alpha \frac{P_k^\alpha}{y_k^\alpha} \cdot y_k^\alpha (\delta_{k,i} - y_i^\alpha) \cdot x_j \\
&= \sum_\alpha \sum_k P_k^\alpha (\delta_{k,i} - y_i^\alpha) x_j \\
&= \sum_\alpha \left(P_i^\alpha - y_i^\alpha \sum_k P_k^\alpha \right) x_j \\
&= \sum_\alpha \underbrace{(P_i^\alpha - y_i^\alpha)}_{\varepsilon^\alpha} x_j
\end{aligned}
$$

$$\frac{\partial H}{\partial y_k} = -\sum_\alpha \frac{P_k^\alpha}{y_k^\alpha}$$

$$\frac{\partial y_k}{\partial z_i} = y_k(\delta_{ki} - y_i)$$

$$\frac{\partial z_i}{\partial w_{ij}} = x_j$$

Gradient of Cross-Entropy + Soft-Max

$$-\frac{\partial H}{\partial \ldots} = \sum - \frac{\partial H}{\partial \ldots} \cdot \frac{\partial y_k}{\partial \ldots} \cdot \frac{\partial z_i}{\partial \ldots}$$

Conclusion

This is the same gradient obtained from minimizing MSE $\sum_\alpha \sum_k (P_k^\alpha - y_i^\alpha)^2$ assuming linear outputs: $\mathbf{y} = \mathbf{z}$

$$\ldots = -\sum_\alpha \frac{P_k^\alpha}{y_k^\alpha}$$

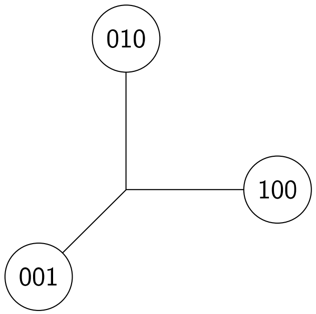
$$\ldots = y_k(\delta_{ki} - y_i)$$

$$\frac{\ldots}{\partial w_{ij}} = x_j$$

$$= \sum_\alpha \underbrace{(P_i^\alpha - y_i^\alpha)}_{\varepsilon^\alpha} x_j$$

Multi-Class Classification

- Use soft-max output layer
- Use cross-entropy cost
- Use one-hot encoding for target values $\mathbf{y}^\alpha$:
  $\mathbf{y}^\alpha = (0,\ldots, 0, 1, 0, \ldots, 0)^t$
  - Ensures equal distance of all class prototypes on hyper cube



Hinge Loss – Optimizing inequalities

- Hinge loss optimizes for inequalities:
  $\forall\, i \neq i^* \quad f_{i^*}(\mathbf{x}) > f_i(\mathbf{x}) + \Delta$
- Hinge loss: $J(i^*) = \sum_{i \neq i^*} \max(0, f_i(\mathbf{x}) - f_{i^*}(\mathbf{x}) + \Delta)$
- Hinge loss becomes zero,
  if inequality constraint is satisfied with margin Δ
- Otherwise cost increases linearly.
- Typical use case: max-margin classifiers, e.g. SVM

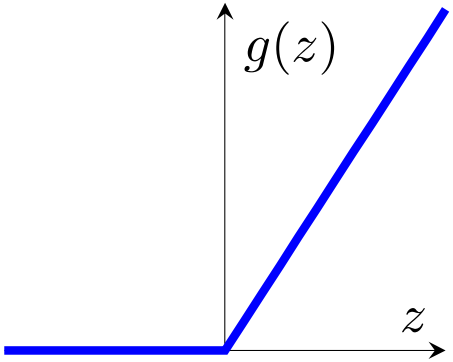
Hinge Loss – Calculation Example

cat	**3.2**	1.3	2.2
car	5.1	**4.9**	2.5
frog	-1.7	2.0	-3.1
loss	1.9	0.0	10.9

Hidden Units

- $\mathbf{h} = \mathrm{g}(\mathbf{z} = \mathrm{W}\,\mathbf{x} + \mathbf{b})$
- default non-linearity g: **Rectified Linear Unit** (ReLu)
  $\mathrm{g}(\mathrm{z}) = \max\,(0, \mathrm{z})$
- ReLu units do not learn
  if activation becomes zero
- *initially* avoid saturation ($\mathrm{z} < 0$)
  $\rightarrow$ use small positive $\mathbf{b} \approx 0.1$

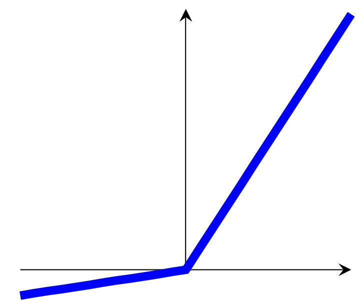


Generalizations of ReLu

- Leaky ReLu

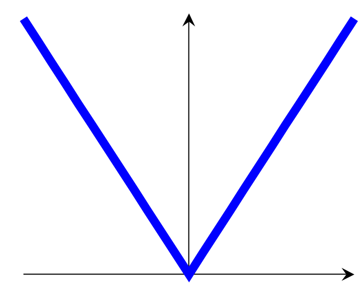
  $g(z) = \max(0, z) + \alpha \min(0, z)$

  with $\alpha \approx 0.01$



- Absolute Value Rectification

  $g(z) = \max(0, z) - \min(0, z)$



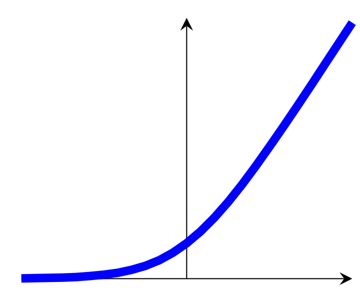
- Parametric ReLu

  $g(z) = \max(0, z) + \alpha \min(0, z)$

  with *trainable* α

- Soft-Plus

  $\zeta(z) = \log(1+\exp(z))$



Linear Hidden Units

- Composing linear functions yields a linear function again.
- Usually, it‘s not useful to stack linear layers.
- They could be squashed into one: $W = U \cdot V$
- Exception: encoder
  - $U \cdot V$: $(p + q) \cdot n$ parameters
  - W: $p \cdot q$ parameters

$$\boxed{p} \xrightarrow{U} \boxed{n} \xrightarrow{V} \boxed{q}$$

Parameter Initialization

- Gradient descent strongly depends on initial params
- Converging at all ?
- Converging how fast ?
- Converging to small training / generalization error ?

Parameter Initialization

- Gradient descent strongly depends on initial params
- Converging at all ?
- Converging how fast ?
- Converging to small training / generalization error ?

- main concern: **breaking symmetry**
  Identically initialized units receiving identical inputs will evolve identically.

→ random initialization

Random Weight Initialization

- Gaussian or Uniform Distribution
  - mean zero
  - main paramater: scale
- large weights

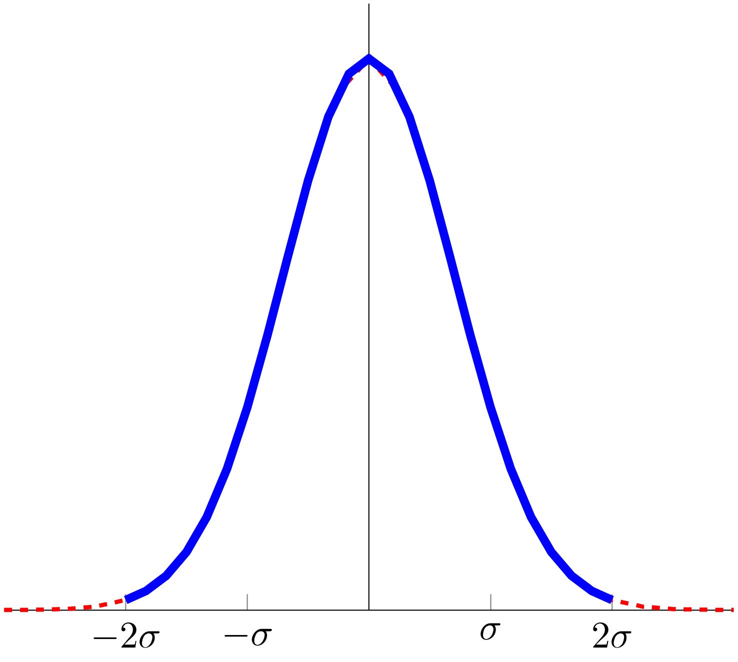
  + better break symmetry

  + avoid losing forward signal

  - risk exploding forward signal

  - take long to be corrected

➔ clip initial weights to 2σ



Xavier Initialization

- Heuristic goal: same activation or gradient variance

- $$w_{ij} \sim \mathcal{N}\left(0, \sigma = \sqrt{\frac{s}{n}}\right)$$
  $$w_{ij} \sim \mathcal{U}\left(-\sqrt{\frac{3s}{n}}, \sqrt{\frac{3s}{n}}\right)$$

Keras Initializers

- Different authors suggest different scale s and n
  - Glorot & Bengio 2010: $s=2$, $n=\mathrm{fan}_{\mathrm{in}} + \mathrm{fan}_{\mathrm{out}}$
  - He 2015: $s=2$, $n=\mathrm{fan}_{\mathrm{in}}$

Weight Scaling - Motivation

$$\mathrm{Var}(\mathbf{w}\cdot\mathbf{x}) = \mathrm{Var}\left(\sum_i^n w_i x_i\right) = \sum_i^n \mathrm{Var}(w_i x_i)$$

for independent random vars w_i, x_i:

$$= \sum_i^n \mathrm{Var}(w_i)\,\mathrm{Var}(x_i) - \mathbb{E}[w_i]^2\mathbb{E}[x_i]^2$$

$$= \sum_i^n \mathrm{Var}(w_i)\,\mathrm{Var}(x_i) = n\,\mathrm{Var}(w_i)\,\mathrm{Var}(x_i)$$

Weight Scaling - Motivation

$$\mathrm{Var}(\mathbf{w}\cdot\mathbf{x}) = \mathrm{Var}\left(\sum_i^n w_i x_i\right) = \sum_i^n \mathrm{Var}(w_i x_i)$$

for independent random vars w_i, x_i:

$$= \sum_i^n \mathrm{Var}(w_i)\,\mathrm{Var}(x_i) - \mathbb{E}[w_i]^2\mathbb{E}[x_i]^2$$

$$= \sum_i^n \mathrm{Var}(w_i)\,\mathrm{Var}(x_i) = n\,\mathrm{Var}(w_i)\,\mathrm{Var}(x_i)$$

Keep variance fixed $\rightarrow$ normalize std. dev. with $\sqrt{n}$

Bias Initialization

- $\mathbf{b} = 0$
- $\mathbf{b} = 0.1$ to ensure initial activation of ReLu units
- For output units in classification tasks
  - bias towards a-priori distribution $\mathrm{p}(\mathbf{y})$
  - $\text{soft-max}(\mathbf{b}) = \mathrm{p}(\mathbf{y})$

Recap: Machine Learning Theory

- Minimize expected costs on a data distribution p(x,y)!

$$J = \mathbb{E}_{p(\mathbf{x},\mathbf{y})}[L(\hat{\mathbf{y}}(\mathbf{x}), \mathbf{y})]$$

- Learning can only optimize w.r.t. a finite training set $\{x^\alpha, y^\alpha\}$ which yields training error

$$J_{\text{train}} = \frac{1}{M_{\text{train}}} \sum_{\alpha=1}^{M_{\text{train}}} L(\hat{\mathbf{y}}(\mathbf{x}^\alpha), \mathbf{y}^\alpha)$$

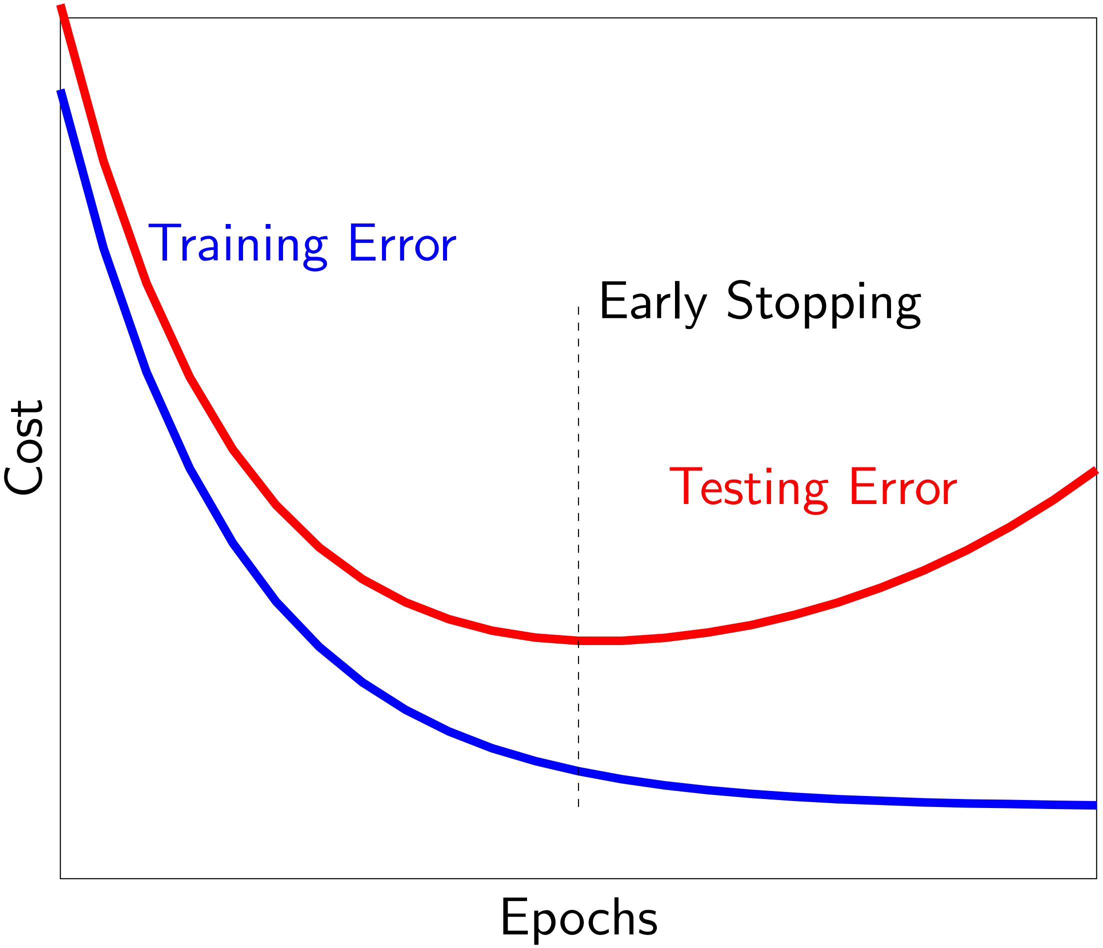
- Estimate generalization error on an *independent* test set

$$J_{\text{test}} = \frac{1}{M_{\text{test}}} \sum_{\alpha=1}^{M_{\text{test}}} L(\hat{\mathbf{y}}(\mathbf{x}^\alpha), \mathbf{y}^\alpha)$$

Training vs. Testing Error

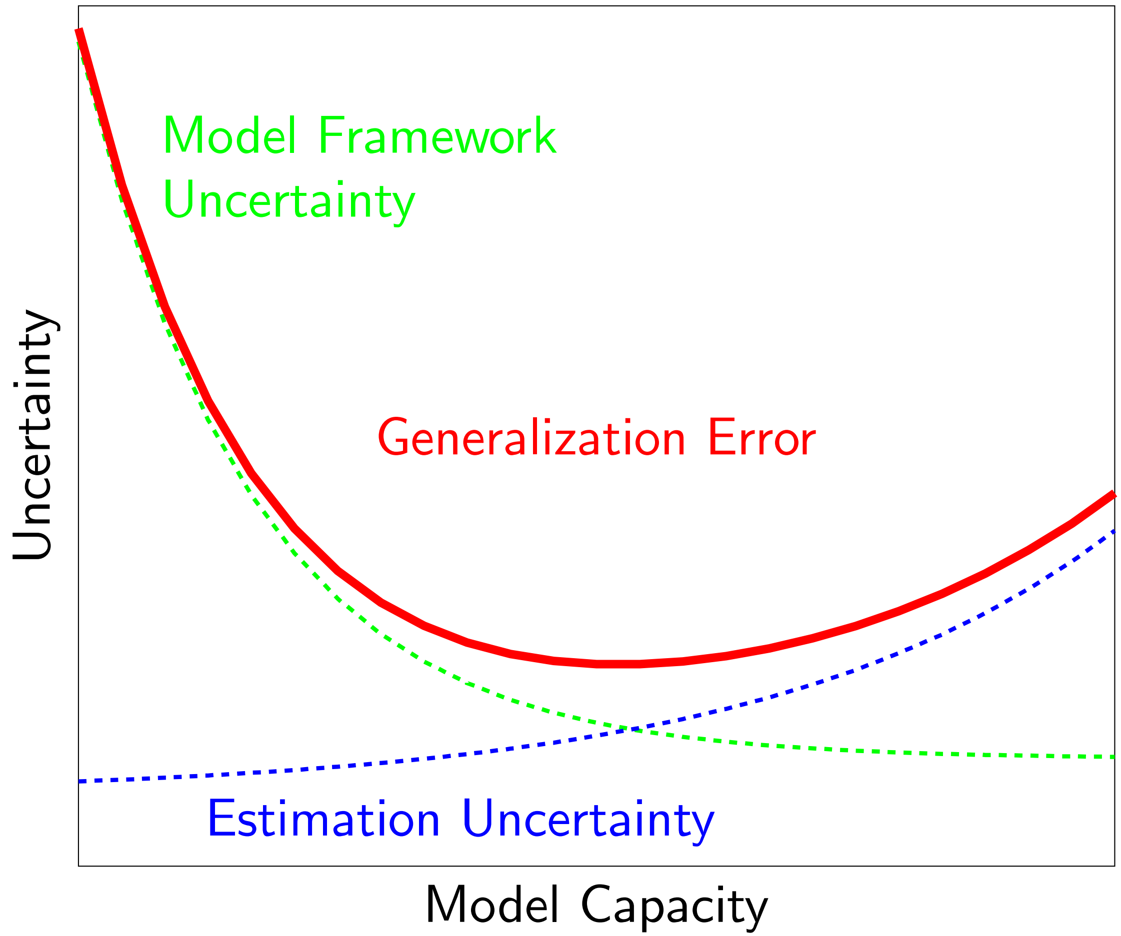
- training error is optimized by learning algorithm
- testing error is usually larger
- testing error usually increases after excessive learning

➔ Early Stopping



Recap: Model Capacity

- model framework uncertainty: insufficiency of model
- estimation uncertainty: insufficiency of data to estimate available model parameters
- estimation uncertainty can be reduced with more training data



Bias – Variance – Dilemma

- Consider data-generating model $f(x, \theta^*)$
- θ^* is true underlying parameter set
- Learner estimates parameter θ, probably a different one each time
- What happens on average?
  - How large is the bias, i.e. estimation error $\theta-\theta^*$?
  - How large is the variance of estimation?
- Both should be small!

Bias – Variance – Dilemma

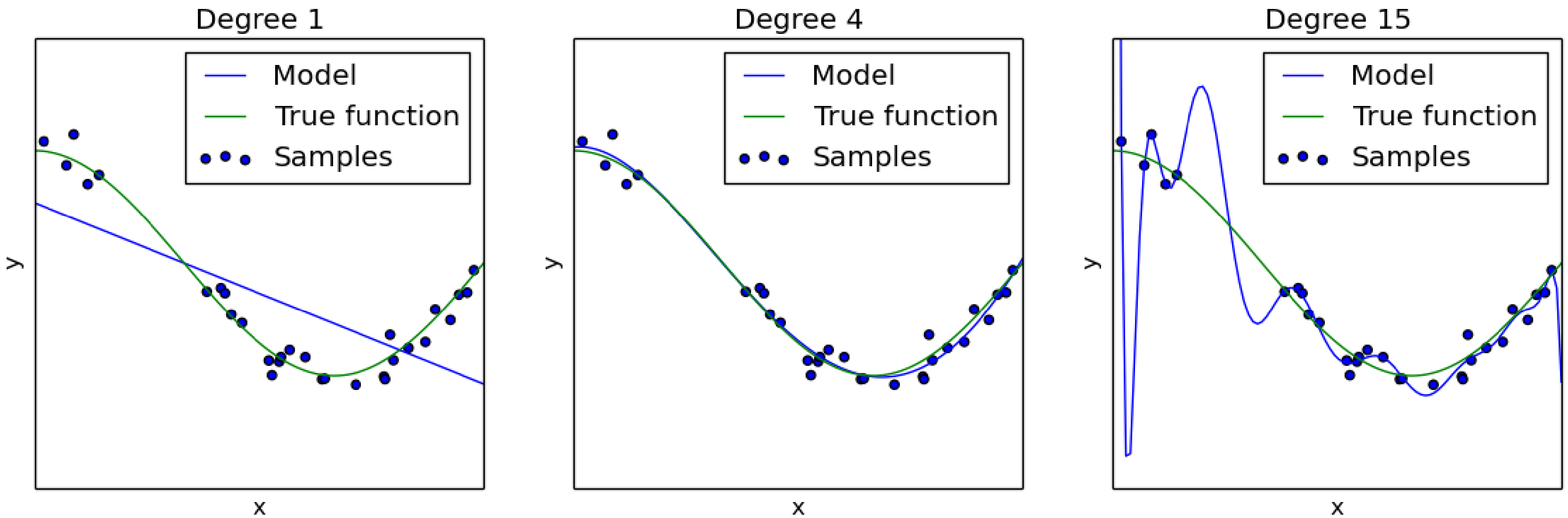
- Let's consider the expected quadratic estimation error

$$\left\langle (\boldsymbol{\theta} - \boldsymbol{\theta}^*)^2 \right\rangle = \left\langle [(\boldsymbol{\theta} - \langle \boldsymbol{\theta} \rangle) + (\langle \boldsymbol{\theta} \rangle - \boldsymbol{\theta}^*)]^2 \right\rangle$$
$$= \left\langle (\boldsymbol{\theta} - \langle \boldsymbol{\theta} \rangle)^2 \right\rangle + (\langle \boldsymbol{\theta} \rangle - \boldsymbol{\theta}^*)^2$$
$$+ 2 \underbrace{\langle \boldsymbol{\theta} - \langle \boldsymbol{\theta} \rangle \rangle}_{0} \cdot (\langle \boldsymbol{\theta} \rangle - \boldsymbol{\theta}^*)$$
$$= \mathsf{Variance}(\boldsymbol{\theta}) + \mathsf{Bias}^2$$

- We cannot reduce both, variance and bias!
- $\nearrow$ model complexity: $\searrow$ bias, $\nearrow$ variance

Example: Model Capacity

- Example: Polynomial Function Fitting
- Model Capacity = Degree of Polynomial
  - too small capacity: under-fitting
  - too high capacity: over-fitting



Regularization

- Deep Networks have high capacity (many parameters)
- Regularization counteracts over-fitting
- … enforcing a simpler model if possible

Regularization

- Deep Networks have high capacity (many parameters)
- Regularization counteracts over-fitting
- … enforcing a simpler model if possible
- Occam‘s Razor (13th century):
  *„If multiple models explain our observations,
  prefer the simplest theory.“*

Regularization

- Deep Networks have high capacity (many parameters)
- Regularization counteracts over-fitting
- … enforcing a simpler model if possible
- Occam‘s Razor (13th century):
  „If multiple models explain our observations,
  prefer the simplest theory.“
- Regularization usually increases the training error,
  but should decrease the generalization error.

Hyper-Parameters

- Hyper-Parameters are model parameters that are not directly optimized by the learning algorithm
  - model capacity
    - polynomial degree
    - network structure, number of units
    - CNNs: filter size + number
  - initialization parameters
  - layer parameters
    - dropout parameter p
    - CNNs: stride, pooling size, ...

Hyper-Parameters

- Hyper-Parameters are model parameters that are not directly optimized by the learning algorithm
- Optimize with Meta-Optimization
  - Grid Search
  - Random Search
- **Do not use test set for hyper-parameter optimization!** Test set is used for independent estimation of test error

→ Split data set into

training set	validation set	test set

Cross-Validation

- Splitting into training, validation, and test set requires huge amounts of data for reasonable statistics

- If only few data is available, use k-fold **cross validation**
- Partition data into k sub sets

1	2	3	4	5	…	k

Cross-Validation

- Splitting into training, validation, and test set requires huge amounts of data for reasonable statistics

- If only few data is available, use k-fold **cross validation**
- Partition data into k sub sets

- k-fold learning, using each sub set for testing once
- Average all test errors

Cross-Validation

- Splitting into training, validation, and test set requires huge amounts of data for reasonable statistics

- If only few data is available, use k-fold **cross validation**
- Partition data into k sub sets

1	2	3	4	5	…	k

- k-fold learning, using each sub set for testing once
- Average all test errors

Cross-Validation

- Splitting into training, validation, and test set requires huge amounts of data for reasonable statistics

- If only few data is available, use k-fold **cross validation**
- Partition data into k sub sets

1	2	3	4	5	…	k

- k-fold learning, using each sub set for testing once
- Average all test errors

Cross-Validation

- Splitting into training, validation, and test set requires huge amounts of data for reasonable statistics

- If only few data is available, use k-fold **cross validation**
- Partition data into k sub sets

1	2	3	4	5	…	k

- k-fold learning, using each sub set for testing once
- Average all test errors

Cross-Validation

- Splitting into training, validation, and test set requires huge amounts of data for reasonable statistics

- If only few data is available, use k-fold **cross validation**
- Partition data into k sub sets

1	2	3	4	5	…	k

- k-fold learning, using each sub set for testing once
- Average all test errors

Cross-Validation

- Splitting into training, validation, and test set requires huge amounts of data for reasonable statistics

- If only few data is available, use k-fold **cross validation**
- Partition data into k sub sets

- k-fold learning, using each sub set for testing once
- Average all test errors

Regularization Methods for Neural Networks

- Weight Norm Penalties: L_2, L_1
- Weight Constraints
- Early Stopping
- Representational Sparsity
- Data Augmentation + Input Noise
- Representational Robustness: Dropout

L_2, L_1 regularization

- Prefer smaller (multiplicative) weights
- Not used for bias b $(\mathrm{z} = \mathrm{W}\ \mathrm{x} + \mathrm{b})$
- Approach: extra cost term $\tilde{J} = J + \lambda\|\mathbf{w}\|_p$

L_2 norm: $\|\mathbf{w}\|_2$

- $\tilde{J} = J + \frac{\lambda}{2}\mathbf{w}^t\mathbf{w}$
- update rule: $\Delta\mathbf{w} = -\eta(\nabla_{\mathbf{w}}J + \lambda\mathbf{w})$

$\rightarrow$ proportional decrease

L_1 norm: $\|\mathbf{w}\|_1$

- $\tilde{J} = J + \lambda\sum_i |w_i|$
- update rule: $\Delta\mathbf{w} = -\eta\,(\nabla_{\mathbf{w}}J + \lambda\,\mathrm{sign}(\mathbf{w}))$

$\rightarrow$ fixed decrease

L_2 regularization

- How this modifies the optimum $\mathbf{w}^*$?
- Let's consider the Taylor expansion of $\tilde{J}$ around $\mathbf{w}^*$:

$$\hat{J}(\mathbf{w}) \equiv \tilde{J}(\mathbf{w}^*) + (\cancel{\nabla_{\mathbf{w}} J} + \lambda \mathbf{w}^*)(\mathbf{w} - \mathbf{w}^*) \\ + \frac{1}{2}(\mathbf{w} - \mathbf{w}^*)^t (H + \lambda \mathbf{1})(\mathbf{w} - \mathbf{w}^*)$$

- $\nabla_{\mathbf{w}} J = 0$ and H are gradient and Hessian of J evaluated at $\mathbf{w}^*$

L_2 regularization

- How this modifies the optimum $\mathbf{w}^*$?
- Let's consider the Taylor expansion of $\tilde{J}$ around $\mathbf{w}^*$:

$$\hat{J}(\mathbf{w}) \equiv \tilde{J}(\mathbf{w}^*) + (\cancel{\nabla_{\mathbf{w}} J} + \lambda \mathbf{w}^*)(\mathbf{w} - \mathbf{w}^*) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^*)^t (H + \lambda \mathbf{1})(\mathbf{w} - \mathbf{w}^*)$$

- $\nabla_{\mathbf{w}} J = 0$ and H are gradient and Hessian of J evaluated at $\mathbf{w}^*$
- New minimum $\tilde{\mathbf{w}}$ for $\nabla_{\mathbf{w}} \hat{J}(\mathbf{w}) = 0$:

$$\lambda \mathbf{w}^* + (H + \lambda \mathbf{1})(\tilde{\mathbf{w}} - \mathbf{w}^*) \stackrel{!}{=} 0$$

$$(H + \lambda \mathbf{1})\tilde{\mathbf{w}} = H \mathbf{w}^*$$

$$\tilde{\mathbf{w}} = (H + \lambda \mathbf{1})^{-1} H \mathbf{w}^*$$

L_2 regularization

- How this modifies the optimum $\mathbf{w}^*$?
- $\tilde{\mathbf{w}} = (H + \lambda \mathbf{1})^{-1} H \mathbf{w}^*$
- $\mathbf{w}^*$ is rescaled along the eigenvector directions of H:

$$\tilde{\mathbf{w}} = Q\Lambda(\Lambda + \lambda \mathbf{1})^{-1} Q^t \mathbf{w}^*$$

- projection of $\mathbf{w}^*$ onto i-th eigenvector is scaled by $\frac{\lambda_i}{\lambda_i + \lambda}$
- Example linear regression:

$$J = (X\mathbf{w} - \mathbf{y})^t (X\mathbf{w} - \mathbf{y}) \quad \rightarrow \quad H = X^t X$$

$$\tilde{\mathbf{w}} = (X^t X + \lambda \mathbf{1})^{-1} X^t X \cdot (X^t X)^{-1} X^t \mathbf{y}$$

- L_2 regularization corresponds to MAP with *Gaussian prior*

L_1 regularization

- How this modifies the optimum $\mathbf{w}^*$?
- Let's consider the Taylor expansion of $\tilde{J}$ around $\mathbf{w}^*$:

$$\hat{J}(\mathbf{w}) \equiv J(\mathbf{w}^*) + \lambda \|\mathbf{w}\|_1 + \frac{1}{2}(\mathbf{w} - \mathbf{w}^*)^t H (\mathbf{w} - \mathbf{w}^*)$$

- If Hessian H of J evaluated at $\mathbf{w}^*$ is *diagonal*, we get:

$$\tilde{w}_i = \mathrm{sign}(w_i^*) \max\left(0, |w_i^*| - \frac{\lambda}{H_{ii}}\right)$$

- w_i^* is shifted towards 0 by amount $\frac{\lambda}{H_{ii}}$ and clipped if necessary
- L_1 enforces **sparsity**

L_1 regularization

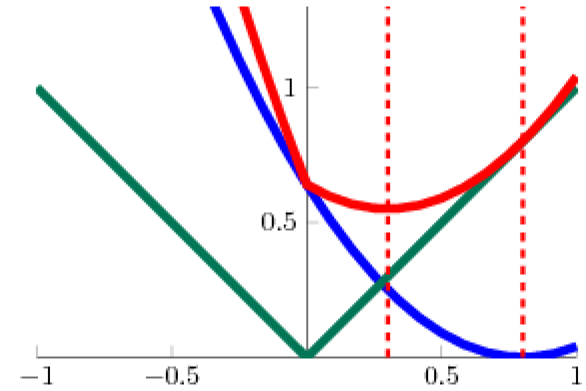
- How this modifies the optimum $\mathbf{w}^*$?
- Let's consider the Taylor expansion of $\tilde{J}$ around $\mathbf{w}^*$:

$$\hat{J}(\mathbf{w}) \equiv J(\mathbf{w}^*) + \lambda \|\mathbf{w}\|_1 + \frac{1}{2}(\mathbf{w} - \mathbf{w}^*)^t H(\mathbf{w} - \mathbf{w}^*)$$

- If Hessian H of J evaluated at $\mathbf{w}^*$ is *diagonal*, we get:

$$\tilde{w}_i = \operatorname{sign}(w_i^*) \max\left(0, |w_i^*| - \frac{\lambda}{H_{ii}}\right)$$

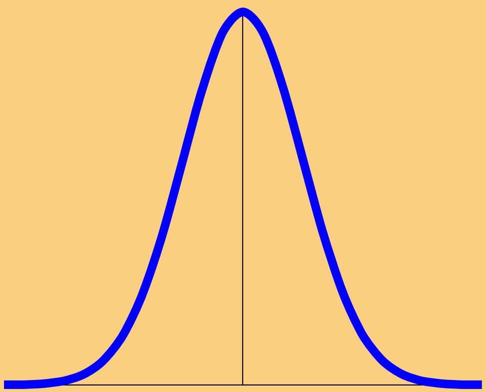
- w_i^* is shifted towards 0 by amount $\frac{\lambda}{H_{ii}}$ and clipped if necessary
- L_1 enforces **sparsity**



L_2, L_1 regularization

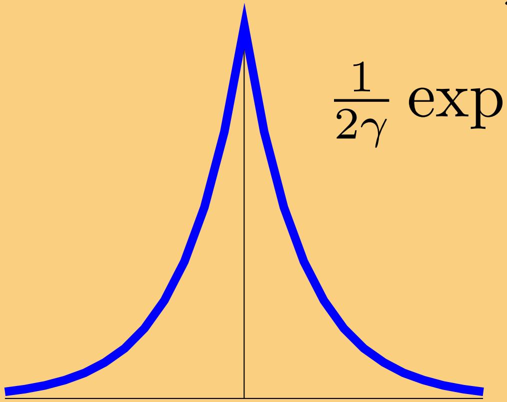
L_2 norm: $\|\mathbf{w}\|_2$

- $\tilde{J} = J + \frac{\lambda}{2}\mathbf{w}^t\mathbf{w}$
- $\Delta\mathbf{w} = -\eta(\nabla_{\mathbf{w}}J + \lambda\mathbf{w})$
- Gaussian prior $\mathcal{N}(0, \frac{1}{\lambda}\mathbf{1})$



L_1 norm: $\|\mathbf{w}\|_1$

- $\tilde{J} = J + \lambda\sum_i |w_i|$
- $\Delta\mathbf{w} = -\eta\left(\nabla_{\mathbf{w}}J + \lambda\,\mathrm{sign}(\mathbf{w})\right)$
- Laplace prior $\mathcal{L}(0, \frac{1}{\lambda}\mathbf{1})$



$$\frac{1}{2\gamma}\exp\left(-\frac{|x-\mu|}{\gamma}\right)$$

- enforces sparsity

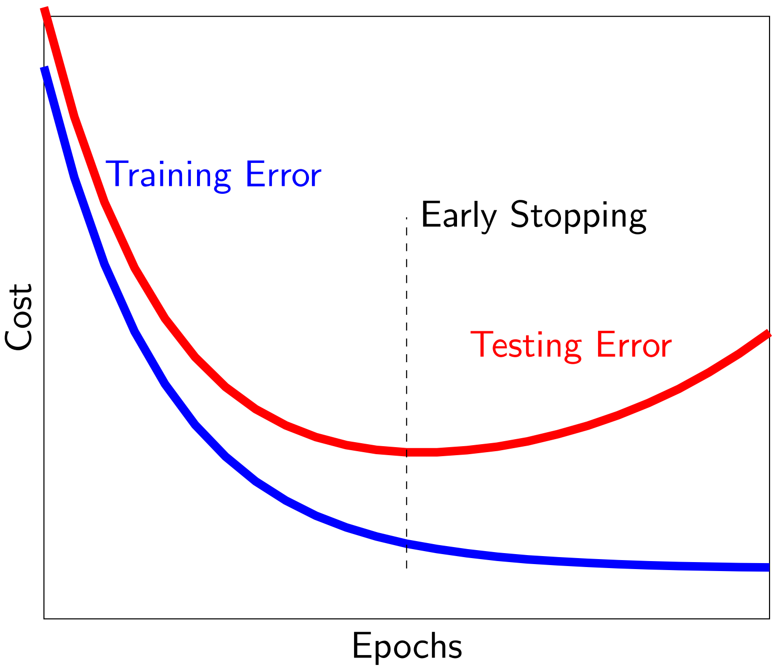
Weight Clipping

- L_2, L_1 regularization smoothly forces weights into unit „ball“ around origin
- Alternatively, explicitly constrain weights: $||\mathbf{w}||_p < \lambda$
  - Take (arbitrary) gradient step
  - *Project* $\mathbf{w}$ back onto constraint, e.g. renormalizing
  - $\lambda \approx$ 3-4
- Allows high learning rate, while avoiding weight explosion
- In practice: constraint norm *for each unit separately*, i.e. row-wise in $\mathrm{Wx+b}$

Keras constraints

Early Stopping

- Monitor validation error (estimating the test error)
- Stop, when it doesn‘t decrease for several steps (and return stored optimal parameters)



- Restricts weight trajectory to stay close to initial values
- Similar effect as L_2 / L_1 regularization
- no extra cost

Representational Sparsity

- L_1 regularization enforced sparse *weights*
- To enforce sparse *representations* (in hidden layers) use an L_1 regularization for hidden layer activations $\mathbf{h}$:

$$\tilde{J} = J + \lambda ||\mathbf{h}||_1$$

Data Augmentation

- Training on more data always improves generalization
- Artificially augment your training set
  - Adding Gaussian noise to the *input* or *hidden layers*
    - Prefers *robust* minima
  - Apply invariant transformations (for classification tasks)
    - shift, rotate, scale, lighten/darken, ... images
    - distort / warp speech signals
    - Be careful not to change class semantics: 6 → 9
- Noise in output layer is *not* useful

Label Smoothing: Noise on class labels

- Classification is trained with
  - soft-max output,
  - cross-entropy cost,
  - and one-hot target vectors $(0,1,0\ldots0)^t$
- Soft-max can never exactly reach zero / one

➔ Training increases weights forever

- Use smoothed one-hot targets:

$$\left(\frac{\varepsilon}{n-1}, 1-\varepsilon, \frac{\varepsilon}{n-1}, \ldots, \frac{\varepsilon}{n-1}\right)^t$$

Ensemble Methods

- Averaging over multiple learners reduces expected variance

$$\begin{aligned}\mathbb{E}\left[\left(\frac{1}{k}\sum_i \epsilon_i\right)^2\right] &= \frac{1}{k^2}\mathbb{E}\left[\sum_i\left(\epsilon_i^2 + \sum_{j\neq i}\epsilon_i\epsilon_j\right)\right] \\ &= \frac{1}{k}v + \frac{k-1}{k}c.\end{aligned}$$

Ensemble Methods

- Averaging over multiple learners reduces expected variance

$$\mathbb{E}\left[\left(\frac{1}{k}\sum_i \epsilon_i\right)^2\right] = \frac{1}{k^2}\mathbb{E}\left[\sum_i\left(\epsilon_i^2 + \sum_{j\neq i}\epsilon_i\epsilon_j\right)\right]$$
$$= \frac{1}{k}v + \frac{k-1}{k}c.$$

- Bagging
  - *same* model trained on *different* datasets
  - same neural network converges to different params
    due to random initialization, random mini-batching, dropout, …
  - computational costs increase linearly with ensemble size

Dropout

- during training: *zero the activity* of randomly selected (prob. p) hidden units (or inputs) *within a mini batch*
  - $\mathrm{p} \approx 0.5$ hidden units
  - $\mathrm{p} \approx 0.8$ input units

- during inference: scale down dropout weights by factor p
- expected total input $\mathrm{pW\,x + b}$ of a unit
  will be similar as in training $\mathrm{W\,px + b}$

Srivastava + Hinton 2014

Effects of Dropout

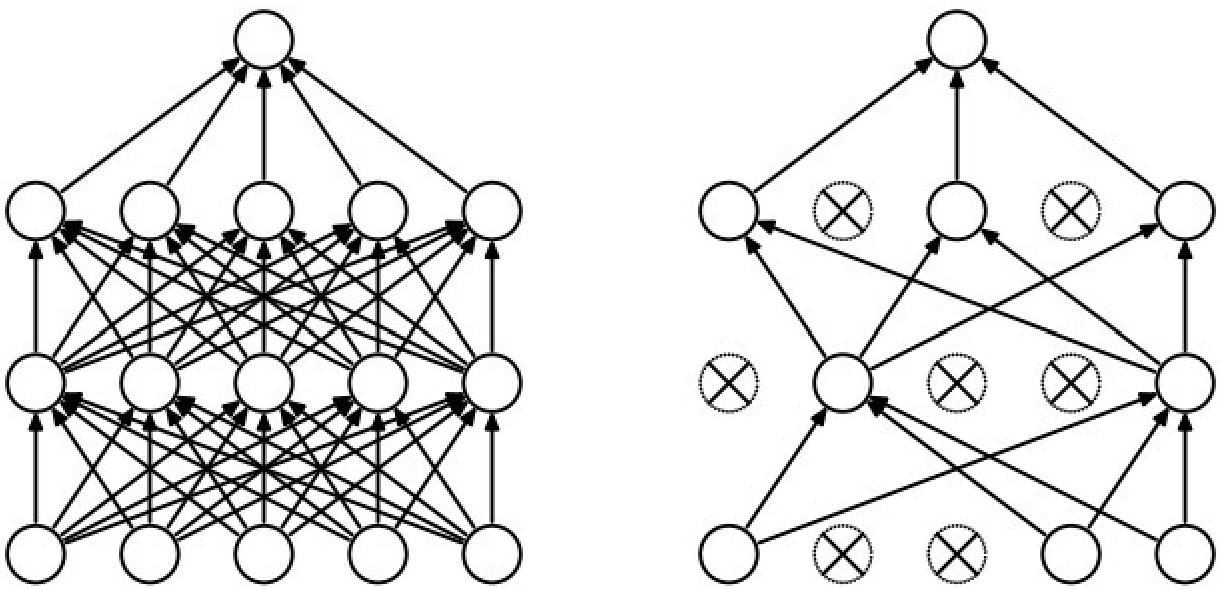
- dropped units do not contribute to output
- other units need to compensate

➔ enforce redundant, distributed representation

- avoid complex co-adaptations of neurons,
  each neuron needs to learn a strong feature on its own
- reduces effective capacity of the network
  → increase layer size
- increases number of epochs by 1/p

Dropout as Ensemble Averaging

- each dropout randomization is new instance of network
- trained on a mini-batch
- sharing weights between all ensemble instances
- full network averages across ensemble



(a) Standard Neural Net (b) After applying dropout.

Batch Normalization

- gradient updates can result in strong output changes

  e.g. in deep linear network:

$$y = x\, w_1\, w_2\, w_3 \cdots w_l$$

$$y' = x\,(w_1 - \varepsilon g_1)\,(w_2 - \varepsilon g_2) \cdots (w_l - \varepsilon g_l)$$

  update has many higher-order terms,

  e.g. $\varepsilon^2 g_1 g_2 \prod_{i=3}^{l} w_i$

- This is due to all layers updated in parallel, while assuming their constness when computing gradients.
- Result: strong change of hidden layer distributions

Batch Normalization

- How to avoid „internal covariate shift“ between updates?
- Naive Approach: Explicit Whitening in mini-batch B

$$\boldsymbol{\mu}_B = \frac{1}{m}\sum_{\alpha=1}^{m} \mathbf{x}_\alpha$$

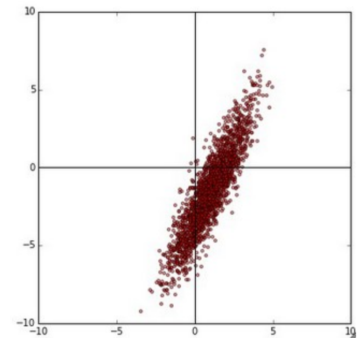
mini-batch mean

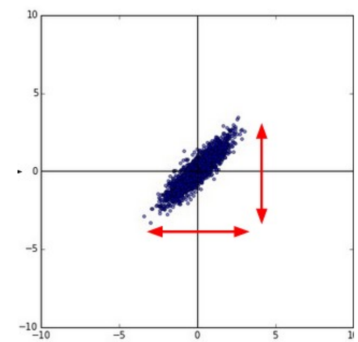
$$\boldsymbol{\sigma}_B^2 = \frac{1}{m}\sum_{\alpha=1}^{m} (\mathbf{x}_\alpha - \boldsymbol{\mu}_B)^2$$

mini-batch variance

$$\hat{\mathbf{x}}_\alpha = \frac{\mathbf{x}_\alpha - \boldsymbol{\mu}_B}{\sqrt{\boldsymbol{\sigma}_B^2 + \varepsilon}}$$

normalization





- *counteracts* progress of gradient descent

Batch Normalization

- New in BN: propagate gradient through normalization
- Ignore gradient components that merely change mean or variance
- Focus on *important* changes

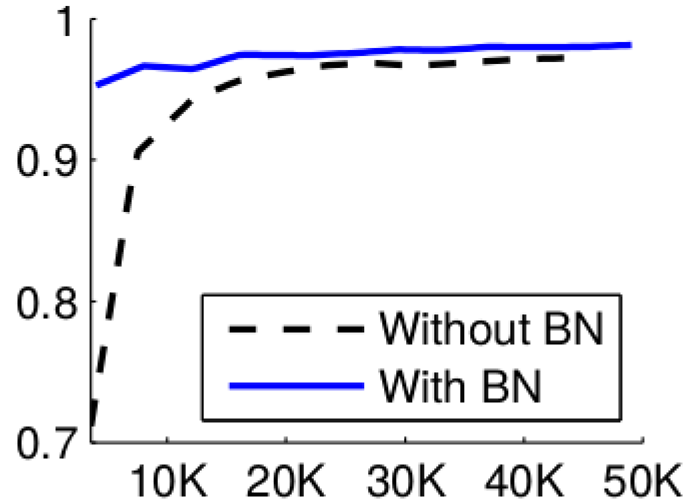
$$\frac{\partial \ell}{\partial \widehat{x}_i} = \frac{\partial \ell}{\partial y_i} \cdot \gamma$$
$$\frac{\partial \ell}{\partial \sigma_\mathcal{B}^2} = \sum_{i=1}^m \frac{\partial \ell}{\partial \widehat{x}_i} \cdot (x_i - \mu_\mathcal{B}) \cdot \frac{-1}{2}(\sigma_\mathcal{B}^2 + \epsilon)^{-3/2}$$
$$\frac{\partial \ell}{\partial \mu_\mathcal{B}} = \left(\sum_{i=1}^m \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{-1}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}} \right) + \frac{\partial \ell}{\partial \sigma_\mathcal{B}^2} \cdot \frac{\sum_{i=1}^m -2(x_i - \mu_\mathcal{B})}{m}$$
$$\frac{\partial \ell}{\partial x_i} = \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{1}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}} + \frac{\partial \ell}{\partial \sigma_\mathcal{B}^2} \cdot \frac{2(x_i - \mu_\mathcal{B})}{m} + \frac{\partial \ell}{\partial \mu_\mathcal{B}} \cdot \frac{1}{m}$$
$$\frac{\partial \ell}{\partial \gamma} = \sum_{i=1}^m \frac{\partial \ell}{\partial y_i} \cdot \widehat{x}_i$$
$$\frac{\partial \ell}{\partial \beta} = \sum_{i=1}^m \frac{\partial \ell}{\partial y_i}$$

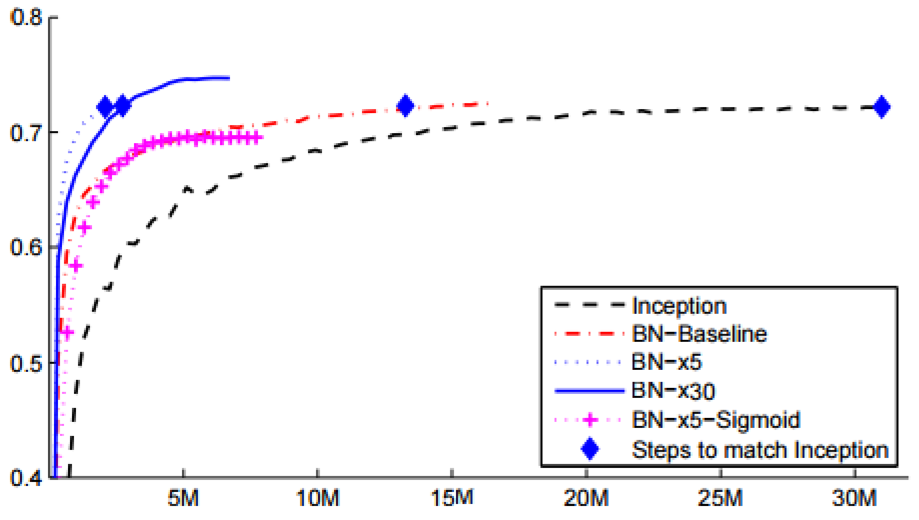
- Restricting mean and variance, restricts model‘s capacity
- Preserve capacity by $\tilde{\mathbf{x}}^\alpha = \boldsymbol{\gamma}\hat{\mathbf{x}}^\alpha + \boldsymbol{\beta}$
  where γ and β are component-wise trainable parameters

Ioffe + Szegedy, 2015

Batch Normalization

- Improves convergence speed
- Allows for higher learning rates
- Regularizing effect, reduces need
  - for dropout
  - for L2 regularization
- Apply *before* nonlinearity, i.e. on $\mathrm{W}\,\mathbf{x} + \mathbf{b}$



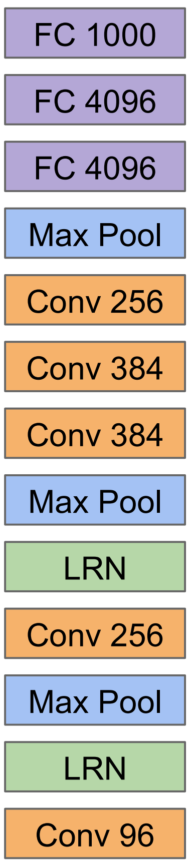


Transfer Learning

- Avoid Learning from scratch
- Start from existing network on a similar task
- Deep Features are often universal,
  i.e. transfer to different tasks

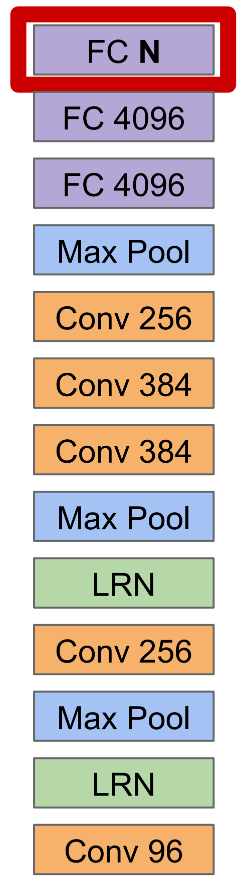
Transfer Learning With Convnets

Find pre-trained network



Small dataset.
Fix all weights (use convnet as feature extractor)
Train only the classifier layer

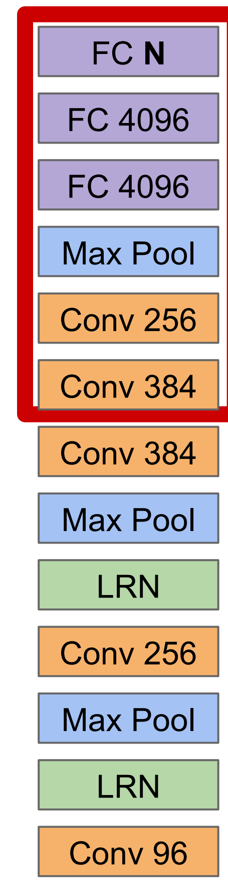
Swap the softmax layer at the end, replace with appropriate number of classes **N**



Medium-sized or larger dataset.
Fine-tune the network.
Use pre-trained net as an initialization.
Train the full network, or just some of the last layers.

Retrain a bigger portion of the network, or even all of it

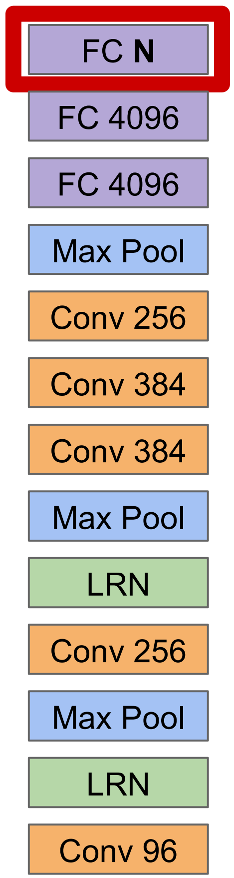
Use only ~1/10th of the original learning rate when fine-tuning the top-most layers and ~1/100th for other layers



A Two-stage Approach to Fine-tuning

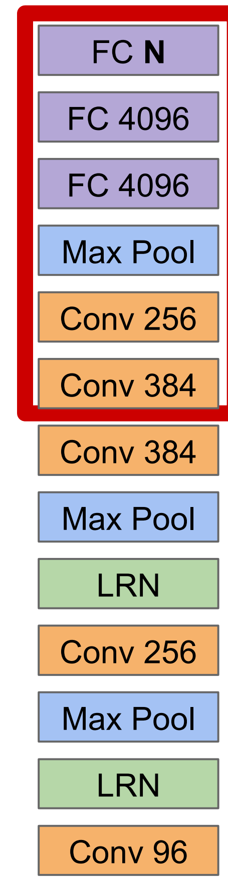
First, train only the new classifier layer to convergence, without modifying the other network weights

Prevents the pre-trained network from being subjected to noisy updates from the randomly initialized classifier layer

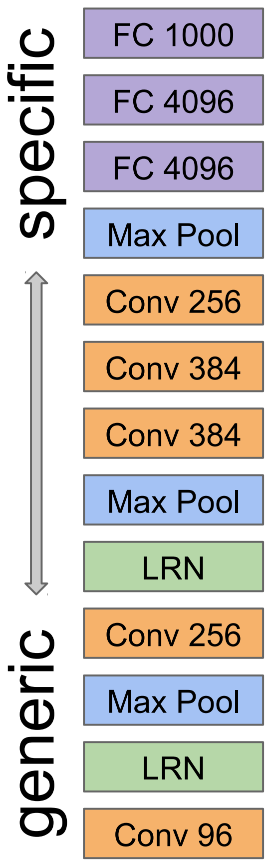


Then, train more of the network, or even all of it (with reduced learning rate)

Fine-tuning can proceed from a good starting point



From the Source to Other Tasks



compared with the source, the **target** dataset is a...

		very similar dataset	**very different dataset**
the **target** dataset has...	**very little data**	Learn linear classifier on features from top layer(s)	You're in trouble … try a linear classifier on different layers
	lots of data	Finetune a few layers	Finetune more layers

Further Reading

- Stanford lecture on Convolutional Networks
- Caffe-in-a-day Tutorial