Learning with Deep Networks: Expressivity, Optimization & Generalization

Charles Ollion - Olivier Grisel

MASTER
DATASCIENCE
UNIVERSITÉ PARIS-SACLAY

Decomposition of the Error

$$E_n(f_n) - E(f^\star) = E(f_{\mathcal{F}}^\star) - E(f^\star) + E_n(f_n) - E(f_{\mathcal{F}}^\star)$$

- Approximation error
- Estimation error

Decomposition of the Error

$$E_n(f_n) - E(f^\star) = E(f_{\mathcal{F}}^\star) - E(f^\star) + E_n(f_n) - E(f_{\mathcal{F}}^\star)$$

- Approximation error
- Estimation error

Ground truth function: $f^\star$

Decomposition of the Error

$$E_n(f_n) - E(f^\star) = E(f_{\mathcal{F}}^\star) - E(f^\star) + E_n(f_n) - E(f_{\mathcal{F}}^\star)$$

- Approximation error
- Estimation error

Ground truth function: $f^\star$

Hypothesis class: $\mathcal{F}$, best hypothesis: $f_{\mathcal{F}}^\star \in \mathcal{F}$

Decomposition of the Error

$$E_n(f_n) - E(f^\star) = E(f_{\mathcal{F}}^\star) - E(f^\star) + E_n(f_n) - E(f_{\mathcal{F}}^\star)$$

- Approximation error
- Estimation error

Ground truth function: $f^\star$

Hypothesis class: $\mathcal{F}$, best hypothesis: $f_{\mathcal{F}}^\star \in \mathcal{F}$

Expected Risk: $E(f) = \int l(f(x), y).\, dP(x, y)$ (but P is unknown)

Decomposition of the Error

$$E_n(f_n) - E(f^\star) = E(f_{\mathcal{F}}^\star) - E(f^\star) + E_n(f_n) - E(f_{\mathcal{F}}^\star)$$

- Approximation error
- Estimation error

Ground truth function: $f^\star$

Hypothesis class: $\mathcal{F}$, best hypothesis: $f_{\mathcal{F}}^\star \in \mathcal{F}$

Expected Risk: $E(f) = \int l(f(x), y).\, dP(x, y)$ (but P is unknown)

Empirical Risk: $E_n(f) = \frac{1}{n} \sum_i l(f(x_i), y_i)$

Decomposition of the Error

$$E_n(f_n) - E(f^\star) = E(f_{\mathcal{F}}^\star) - E(f^\star) + E_n(f_n) - E(f_{\mathcal{F}}^\star)$$

- Approximation error
- Estimation error

Ground truth function: $f^\star$

Hypothesis class: $\mathcal{F}$, best hypothesis: $f_{\mathcal{F}}^\star \in \mathcal{F}$

Expected Risk: $E(f) = \int l(f(x), y).\, dP(x, y)$ (but P is unknown)

Empirical Risk: $E_n(f) = \frac{1}{n} \sum_i l(f(x_i), y_i)$

Empirical Risk Minimization:

$$f_n = argmin_{f \in \mathcal{F}} E_n(f)$$

Decomposition of the Error

$$
\begin{aligned}
E_n(\tilde{f}_n) - E(f^\star) &= E(f_{\mathcal{F}}^\star) - E(f^\star) \\
&+ E_n(f_n) - E(f_{\mathcal{F}}^\star) \\
&+ E_n(\tilde{f}_n) - E_n(f_n)
\end{aligned}
$$

- Approximation error
- Estimation error
- Optimization error

Learning using Large Datasets, L. Bottou & O. Bousquet, 2008.

Decomposition of the Error

$$
\begin{aligned}
E_n(\tilde{f}_n) - E(f^\star) &= E(f_{\mathcal{F}}^\star) - E(f^\star) \\
&+ E_n(f_n) - E(f_{\mathcal{F}}^\star) \\
&+ E_n(\tilde{f}_n) - E_n(f_n)
\end{aligned}
$$

- Approximation error
- Estimation error
- Optimization error

Computing $argmin_{f \in \mathcal{F}} E_n(f)$ exactly can be very costly...

Learning using Large Datasets, L. Bottou & O. Bousquet, 2008.

Decomposition of the Error

$$
\begin{aligned}
E_n(\tilde{f}_n) - E(f^\star) &= E(f_{\mathcal{F}}^\star) - E(f^\star) \\
&+ E_n(f_n) - E(f_{\mathcal{F}}^\star) \\
&+ E_n(\tilde{f}_n) - E_n(f_n)
\end{aligned}
$$

- Approximation error
- Estimation error
- Optimization error

Computing $argmin_{f \in \mathcal{F}} E_n(f)$ exactly can be very costly...

or even untractable e.g. for non-convex objective:
$\theta \rightarrow l(f_\theta(x_i), y_i)$

Learning using Large Datasets, L. Bottou & O. Bousquet, 2008.

Decomposition of the Error

$$
\begin{aligned}
E_n(\tilde{f}_n) - E(f^\star) &= E(f_{\mathcal{F}}^\star) - E(f^\star) \\
&+ E_n(f_n) - E(f_{\mathcal{F}}^\star) \\
&+ E_n(\tilde{f}_n) - E_n(f_n)
\end{aligned}
$$

- Approximation error
- Estimation error
- Optimization error

Computing $argmin_{f \in \mathcal{F}} E_n(f)$ exactly can be very costly...

or even untractable e.g. for non-convex objective:
$\theta \rightarrow l(f_\theta(x_i), y_i)$

In practice: approximate optimizer with tolerance ρ that finds $\tilde{f}_n$:

$$E_n(\tilde{f}_n) < E_n(f_n) + \rho \quad s.t. \quad E(\tilde{f}_n) \approx E(f_n)$$

Learning using Large Datasets, L. Bottou & O. Bousquet, 2008.

Decomposition of the Error

Approximation error:

- decreases when $\mathcal{F}$ increases;
- but typically bounded by computational constraints.

Decomposition of the Error

Approximation error:

- decreases when $\mathcal{F}$ increases;
- but typically bounded by computational constraints.

Estimation error:

- decreases when n increases;
- can increase when $\mathcal{F}$ increases (according to VC theory).

Decomposition of the Error

Approximation error:

- decreases when $\mathcal{F}$ increases;
- but typically bounded by computational constraints.

Estimation error:

- decreases when n increases;
- can increase when $\mathcal{F}$ increases (according to VC theory).

Optimization error:

- can increase when tolerance ρ increases;
- can increase when $\mathcal{F}$ gets more complex (non-convex obj.).

Outline

Approximation

Outline

Approximation

Optimization

Outline

Approximation

Optimization

Estimation

Expressivity and Universal Function Approximation

Universal Function Approximation

Let σ be a nonconstant, bounded, and monotonically-increasing continuous function. For any $f \in C([0,1]^d)$ and $\varepsilon > 0$, there exists $h \in \mathbb{N}$ real constants $v_i, b_i \in \mathbb{R}$ and real vectors $w_i \in \mathbb{R}^d$ such that:

$$|\sum_{i}^{h} v_i \sigma(w_i^T x + b_i) - f(x)| < \varepsilon$$

that is: neural nets are dense in $C([0,1]^d)$.

This still holds for any compact subset of $\mathbb{R}^d$ and if σ is the ReLU function (Sonoda & Murata, 2015).

Approximation Capabilities of Multilayer Feedforward Networks, K. Hornik, 1991

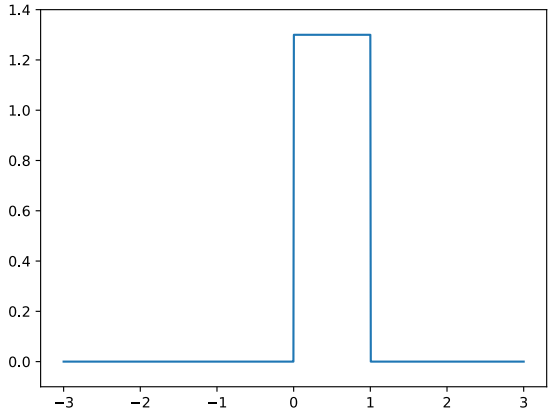
Problem solved?

What UFA theorems **do NOT tell us**:

- The number h of hidden units is small enough to have the network fit in RAM.
- The optimal function parameters can be found in finite time by minimizing the Empirical Risk with SGD and the usual random initialization schemes.

Approximation with ReLU nets

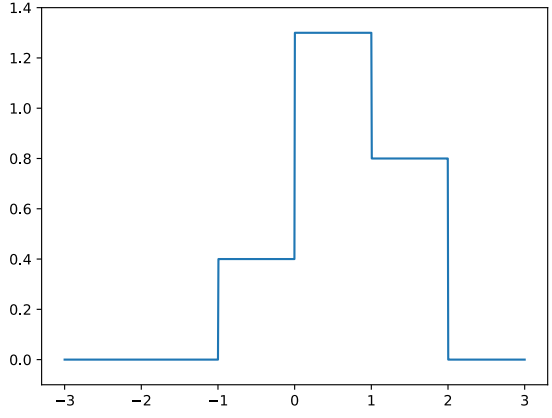
```python
import numpy as np
import matplotlib.pyplot as plt


def relu(x):
    return np.maximum(x, 0)


def rect(x, a, b, h, eps=1e-7):
    return h / eps * (
           relu(x - a)
         - relu(x - (a + eps))
         - relu(x - b)
         + relu(x - (b + eps)))


x = np.linspace(-3, 3, 1000)
y = rect(x, 0, 1, 1.3)

plt.plot(x, y)
```



Quora: Is a single layered ReLU network still a universal approximator?, Conner Davis

Approximation with ReLU nets

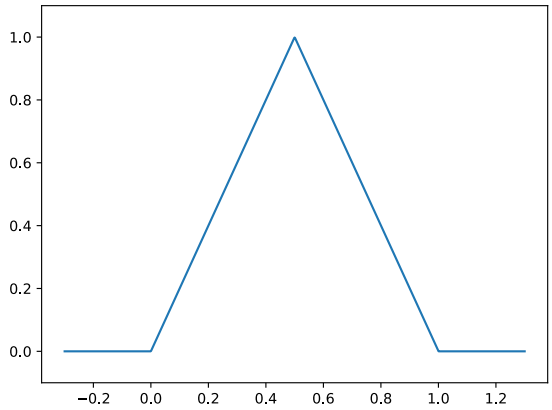
```python
import numpy as np
import matplotlib.pyplot as plt


def relu(x):
    return np.maximum(x, 0)


def rect(x, a, b, h, eps=1e-7):
    return h / eps * (
           relu(x - a)
         - relu(x - (a + eps))
         - relu(x - b)
         + relu(x - (b + eps)))


x = np.linspace(-3, 3, 1000)
y = (  rect(x, -1, 0, 0.4)
     + rect(x,  0, 1, 1.3)
     + rect(x,  1, 2, 0.8))
plt.plot(x, y)
```



Quora: Is a single layered ReLU network still a universal approximator?, Conner Davis

Efficient Oscillations with Composition

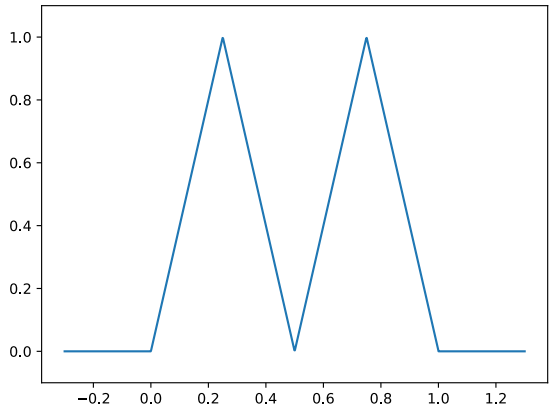
```
import numpy as np
import matplotlib.pyplot as plt


def relu(x):
    return np.maximum(x, 0)


def tri(x):
    return relu(  relu(2 * x)
                - relu(4 * x - 2))


x = np.linspace(-.3, 1.3, 1000)
y = tri(x)

plt.plot(x, y)
```



Benefits of depth in neural networks, Matus Telgarsky, 2016

Efficient Oscillations with Composition

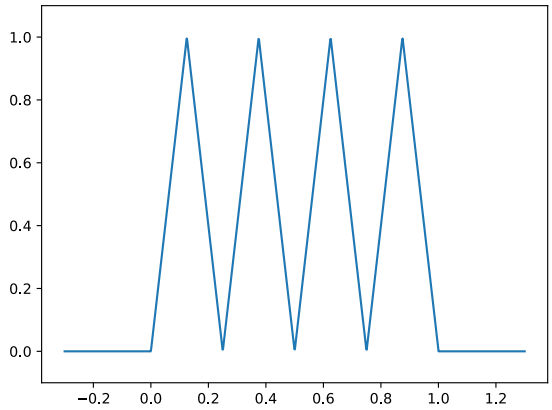
```
import numpy as np
import matplotlib.pyplot as plt


def relu(x):
    return np.maximum(x, 0)


def tri(x):
    return relu(  relu(2 * x)
                - relu(4 * x - 2))


x = np.linspace(-.3, 1.3, 1000)
y = tri(tri(x))

plt.plot(x, y)
```



Benefits of depth in neural networks, Matus Telgarsky, 2016

Efficient Oscillations with Composition

```
import numpy as np
import matplotlib.pyplot as plt


def relu(x):
    return np.maximum(x, 0)


def tri(x):
    return relu(  relu(2 * x)
                - relu(4 * x - 2))


x = np.linspace(-.3, 1.3, 1000)
y = tri(tri(tri(x)))

plt.plot(x, y)
```

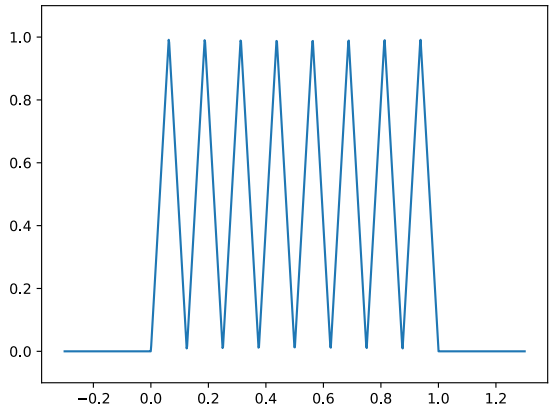


1 more layer → 2x more oscillations

Benefits of depth in neural networks, Matus Telgarsky, 2016

Efficient Oscillations with Composition

```
import numpy as np
import matplotlib.pyplot as plt


def relu(x):
    return np.maximum(x, 0)


def tri(x):
    return relu(  relu(2 * x)
                - relu(4 * x - 2))


x = np.linspace(-.3, 1.3, 1000)
y = tri(tri(tri(tri(x))))

plt.plot(x, y)
```



1 more layer → 2x more oscillations

Benefits of depth in neural networks, Matus Telgarsky, 2016

Depth and Parametric Cost

Matus Telgarsky, 2016: There exists functions that can be approximated by a deep ReLU network with $\Theta(k^3)$ layers with a $\Theta(1)$ units that cannot be approximated by shallower networks with $\Theta(k)$ layers unless they have $\Omega(2^k)$ units.

Note: the number of parameters of a deep network is typically quadratic with the number of units.

For a fixed param budget, deeper is better

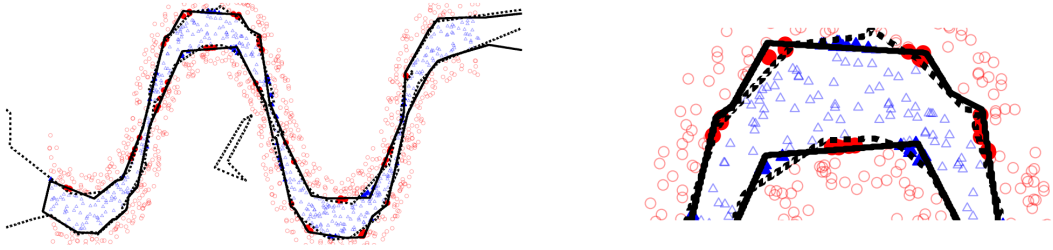


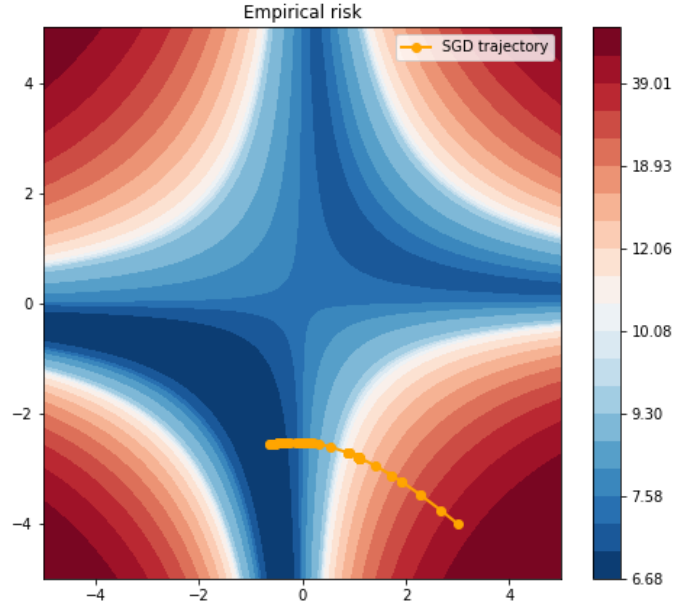
Figure 1: Binary classification using a shallow model with 20 hidden units (solid line) and a deep model with two layers of 10 units each (dashed line). The right panel shows a close-up of the left panel. Filled markers indicate errors made by the shallow model.

On the Number of Linear Regions of Deep Neural Networks, G. Montúfar, R. Pascanu, K. Cho, Y. Bengio, 2014.

Optimization for Deep Networks

MLP Loss

```
import numpy as np
rng = np.random.RandomState(42)


def relu(x):
    return np.maximum(x, 0)


def mlp(x, w1, w2):
    return w2 * relu(w1 * x)


def sq_loss(y, y_hat):
    return (y - y_hat) ** 2


n = 30
x = np.abs(rng.randn(n) + 1)
y = 2 * x + 0.5 * rng.randn(n)

l = 0.
for xi, yi in zip(x, y):
    l += sq_loss(yi - mlp(w1, w2, xi))
l /= len(x)
```



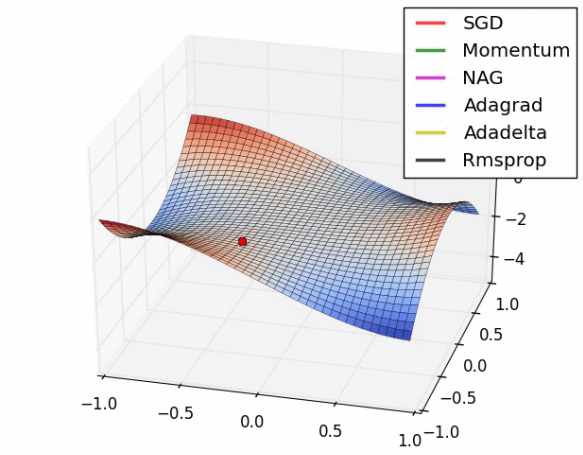
Stochastic MLP Loss

```python
import numpy as np
rng = np.random.RandomState(42)


def relu(x):
    return np.maximum(x, 0)


def mlp(x, w1, w2):
    return w2 * relu(w1 * x)


def sq_loss(y, y_hat):
    return (y - y_hat) ** 2


n = 30
x = np.abs(rng.randn(n) + 1)
y = 2 * x + 0.5 * rng.randn(n)

l = 0.
for xi, yi in zip(x, y):
    l += sq_loss(yi - mlp(w1, w2, xi)
l /= len(x)
```

Non-convex Optimization

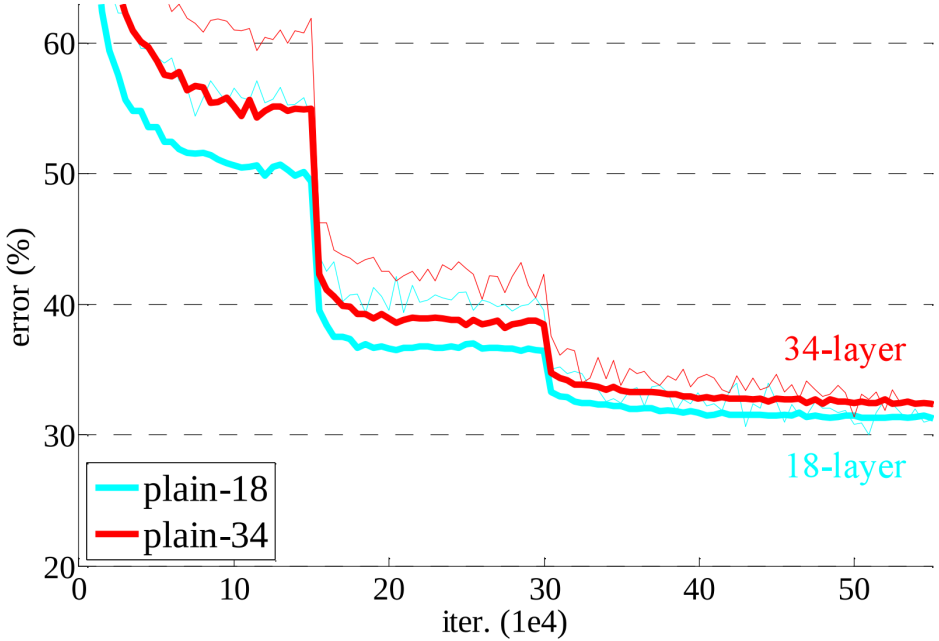
No global minimum convergence guarantee but FF nets are often easy to train:

- Different random seeds yield similar final train losses.
- When wide enough, training error can often reach zero: no bad local minima.
- SGD (with momentum) seem to be able to evade saddle structures quite easily.



Credits: Alec Radford

Training Deeper ConvNets



Deep Residual Learning for Image Recognition, K. He et al., 2015.

Conditioning Issues

Vanishing gradients:

- Deep FF Network: smaller gradient components in first layers;
- Vanilla RNN with many time steps.

Conditioning Issues

Vanishing gradients:

- Deep FF Network: smaller gradient components in first layers;
- Vanilla RNN with many time steps.

Text model with word embeddings as input:

- Some words are much less frequent than others;
- Rare non-zero gradient for matching columns in embedding.

Conditioning Issues

Vanishing gradients:

- Deep FF Network: smaller gradient components in first layers;
- Vanilla RNN with many time steps.

Text model with word embeddings as input:

- Some words are much less frequent than others;
- Rare non-zero gradient for matching columns in embedding.

Optimal learning rate should be higher for some gradient components:

- Bad conditioning of the Hessian matrix.

Fixing Conditioning Issues

Fixing the optimizer:

- AdaGrad, RMSProp, AdaDelta, Adam
  - Diagonal preconditioned stochastic updates
  - Cheap rescaling of the gradient components by the square root of the (moving) average of g_i^2
- Non-diagonal preconditioned stochastic updates: K-FAC & Shampoo

Fixing Conditioning Issues

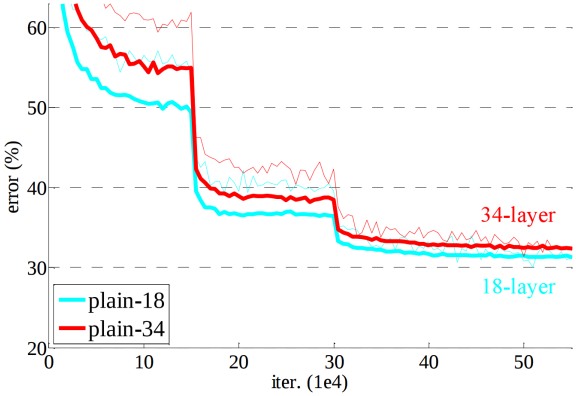
Fixing the optimizer:

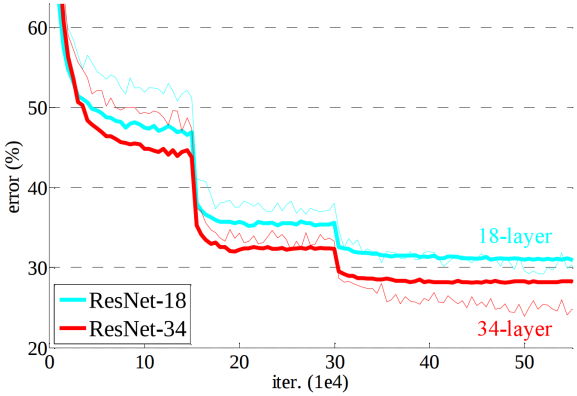
- AdaGrad, RMSProp, AdaDelta, Adam
  - Diagonal preconditioned stochastic updates
  - Cheap rescaling of the gradient components by the square root of the (moving) average of g_i^2
- Non-diagonal preconditioned stochastic updates: K-FAC & Shampoo

Fixing the architecture:

- Skip, Residual connections (e.g. ResNets)

Residual Networks





Deep Residual Learning for Image Recognition, K. He et al., 2015.

Batch Normalization

Normalize activations in each **mini-batch** before activation function: **speeds up** and **stabilizes** training (less dependent on init)

Ioffe, Sergey, and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." ICML 2015

Batch Normalization

Normalize activations in each **mini-batch** before activation function: **speeds up** and **stabilizes** training (less dependent on init)

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Ioffe, Sergey, and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." ICML 2015

Batch Normalization

At **inference time**, use average and standard deviation computed on **the whole dataset** instead of batch

Batch Normalization

At **inference time**, use average and standard deviation computed on **the whole dataset** instead of batch

Widely used in **ConvNets**, but requires the mini-batch to be large enough to compute statistics in the minibatch.

Batch Normalization

At **inference time**, use average and standard deviation computed on **the whole dataset** instead of batch

Widely used in **ConvNets**, but requires the mini-batch to be large enough to compute statistics in the minibatch.

In **Keras**: use it as a layer, before activation

```
x = Convolution2D(64, 1, 1)(input_tensor)
x = BatchNormalization()(x)
x = Activation('relu')(x)
```

Batch Normalization

At **inference time**, use average and standard deviation computed on **the whole dataset** instead of batch

Widely used in **ConvNets**, but requires the mini-batch to be large enough to compute statistics in the minibatch.

In **Keras**: use it as a layer, before activation

```
x = Convolution2D(64, 1, 1)(input_tensor)
x = BatchNormalization()(x)
x = Activation('relu')(x)
```

- Introduces new parameters: `2 x size_of_activation` (here 128)

Batch Normalization

At **inference time**, use average and standard deviation computed on **the whole dataset** instead of batch

Widely used in **ConvNets**, but requires the mini-batch to be large enough to compute statistics in the minibatch.

In **Keras**: use it as a layer, before activation

```
x = Convolution2D(64, 1, 1)(input_tensor)
x = BatchNormalization()(x)
x = Activation('relu')(x)
```

- Introduces new parameters: `2 x size_of_activation` (here 128)
- Keras keeps track of a **running average/std** of activations for inference, no need to have specific inference code.

Batch Normalization

At **inference time**, use average and standard deviation computed on **the whole dataset** instead of batch

Widely used in **ConvNets**, but requires the mini-batch to be large enough to compute statistics in the minibatch.

In **Keras**: use it as a layer, before activation

```
x = Convolution2D(64, 1, 1)(input_tensor)
x = BatchNormalization()(x)
x = Activation('relu')(x)
```

- Introduces new parameters: `2 x size_of_activation` (here 128)
- Keras keeps track of a **running average/std** of activations for inference, no need to have specific inference code.

Layer Normalizations

Normalize on the statistics of the **layer activations** instead of mini-batch.

$$\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l \qquad \sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}\left(a_i^l - \mu^l\right)^2}$$

Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E. Hinton. "Layer normalization." 2016.

Layer Normalizations

Normalize on the statistics of the **layer activations** instead of mini-batch.

$$\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l \qquad \sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}\left(a_i^l - \mu^l\right)^2}$$

The algorithm is then similar as Batch Normalization

Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E. Hinton. "Layer normalization." 2016.

Layer Normalizations

Normalize on the statistics of the **layer activations** instead of mini-batch.

$$\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l \qquad \sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}\left(a_i^l - \mu^l\right)^2}$$

The algorithm is then similar as Batch Normalization

Suited for **RNNs** and **Transformers**, degrades performance of **CNNs**

Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E. Hinton. "Layer normalization." 2016.

Weight Normalization

Reparametrize weights of neurons, to decouple **direction** and **norm** of the weight:

$$\mathbf{w} = \frac{g}{||\mathbf{v}||}\mathbf{v}$$

Salimans, Tim, and Diederik P. Kingma. "Weight normalization: A simple reparameterization to accelerate training of deep neural networks." NIPS 2016.

Weight Normalization

Reparametrize weights of neurons, to decouple **direction** and **norm** of the weight:

$$\mathbf{w} = \frac{g}{||\mathbf{v}||}\mathbf{v}$$

One new parameter g to learn per neuron

Salimans, Tim, and Diederik P. Kingma. "Weight normalization: A simple reparameterization to accelerate training of deep neural networks." NIPS 2016.

Weight Normalization

Reparametrize weights of neurons, to decouple **direction** and **norm** of the weight:

$$\mathbf{w} = \frac{g}{||\mathbf{v}||}\mathbf{v}$$

One new parameter g to learn per neuron

Careful **data-based** initialization of g and neuron bias b is better (not applicable to RNNs)

Salimans, Tim, and Diederik P. Kingma. "Weight normalization: A simple reparameterization to accelerate training of deep neural networks." NIPS 2016.

Weight Normalization

Reparametrize weights of neurons, to decouple **direction** and **norm** of the weight:

$$\mathbf{w} = \frac{g}{||\mathbf{v}||}\mathbf{v}$$

One new parameter g to learn per neuron

Careful **data-based** initialization of g and neuron bias b is better (not applicable to RNNs)

More recently: Weight Standardization

Salimans, Tim, and Diederik P. Kingma. "Weight normalization: A simple reparameterization to accelerate training of deep neural networks." NIPS 2016.

Reparametrizations and Normalizations

Significant impact on results

- Batch Norm makes convergence faster to an optimum that generalizes better on CNNs

Reparametrizations and Normalizations

Significant impact on results

- Batch Norm makes convergence faster to an optimum that generalizes better on CNNs
- Layer Norm is typically used in RNNs and Transformer architectures

Reparametrizations and Normalizations

Significant impact on results

- Batch Norm makes convergence faster to an optimum that generalizes better on CNNs
- Layer Norm is typically used in RNNs and Transformer architectures

- Active area of research:
  - Normalizer-Free ResNets (NFNets) SOTA on ImageNet with Scaled Weight Standardization

BTW: why minibatch SGD?

High capacity Deep Networks require a **large number of training samples** (typically at least 10,000).

BTW: why minibatch SGD?

High capacity Deep Networks require a **large number of training samples** (typically at least 10,000).

Samples contribute **redundant gradient information**. Otherwise generalization would not be possible and no learning would ever happen.

BTW: why minibatch SGD?

High capacity Deep Networks require a **large number of training samples** (typically at least 10,000).

Samples contribute **redundant gradient information**. Otherwise generalization would not be possible and no learning would ever happen.

SGD is more computationally efficient when dealing with redundant samples:

- many updates per-epoch for SGD, one for full-batch GD
- doubling the size of the training with copies of samples would double the training time with full-batch GD
- no impact on training time of SGD.

Generalization

Overfitting?

Using (decoupled) weight decay can help a bit.

Overfitting?

Using (decoupled) weight decay can help a bit.

Reducing width can cause optimization issues.

Overfitting?

Using (decoupled) weight decay can help a bit.

Reducing width can cause optimization issues.

Early stopping can help a bit but mostly useful to avoid wasting compute optimizing for nothing.

Overfitting?

Using (decoupled) weight decay can help a bit.

Reducing width can cause optimization issues.

Early stopping can help a bit but mostly useful to avoid wasting compute optimizing for nothing.

Data augmentation and stochastic regularization (dropout) often help but:

- it also makes training (much) slower,
- too much gradient variance can prevent training completely.

Overfitting?

Using (decoupled) weight decay can help a bit.

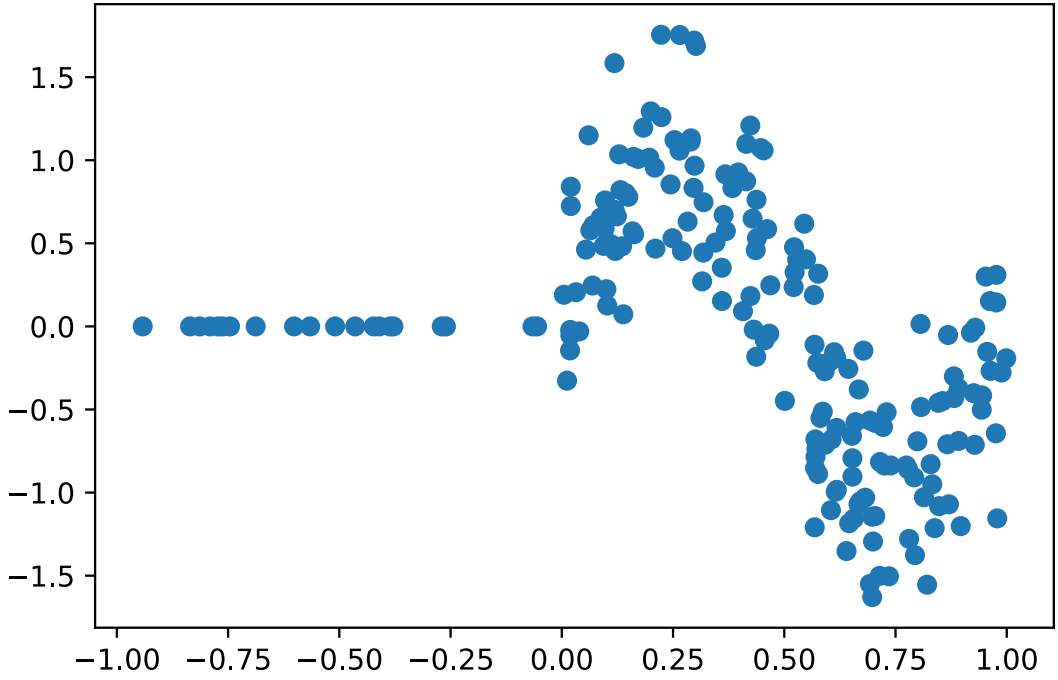
Reducing width can cause optimization issues.

Early stopping can help a bit but mostly useful to avoid wasting compute optimizing for nothing.

Data augmentation and stochastic regularization (dropout) often help but:

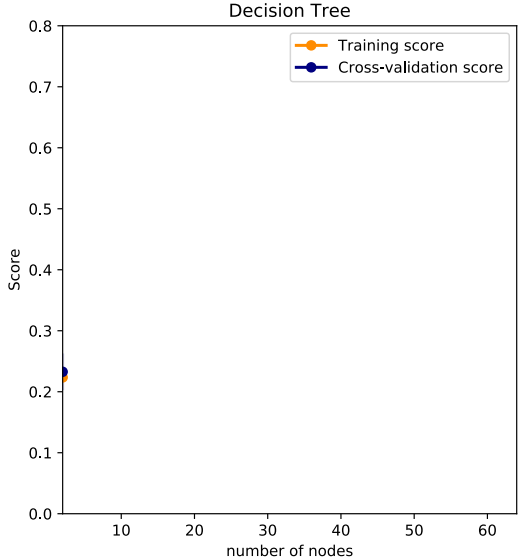
- it also makes training (much) slower,
- too much gradient variance can prevent training completely.

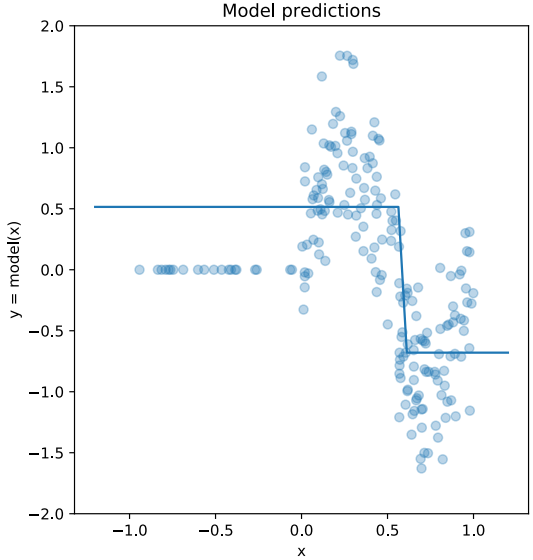
Ensembling models can help and can be reasonably cheap with SGD restarts ("snapshot ensembles").



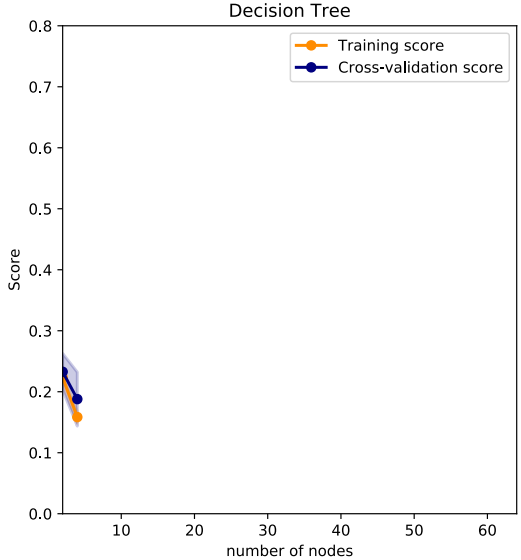
Experiment: regression on 1D data with heteroschedastic noise

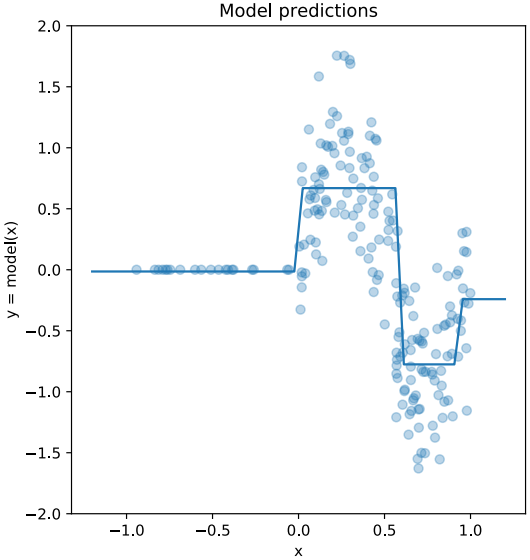
Single Decision Tree



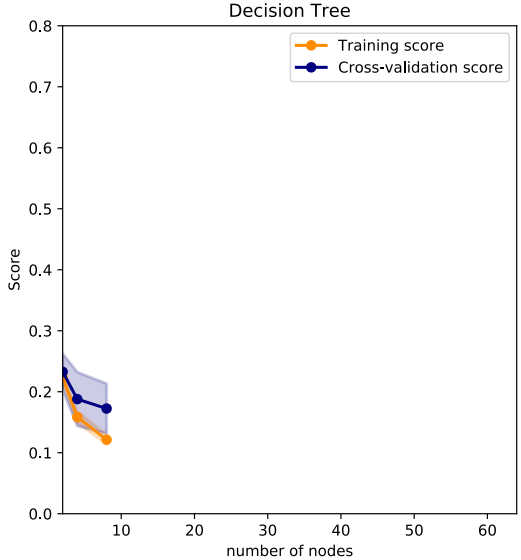


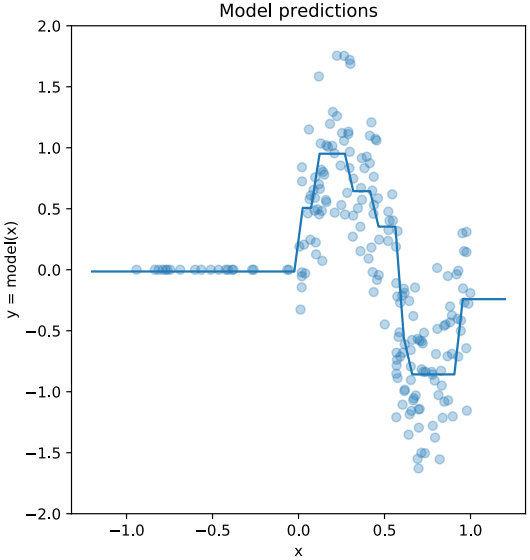
Single Decision Tree



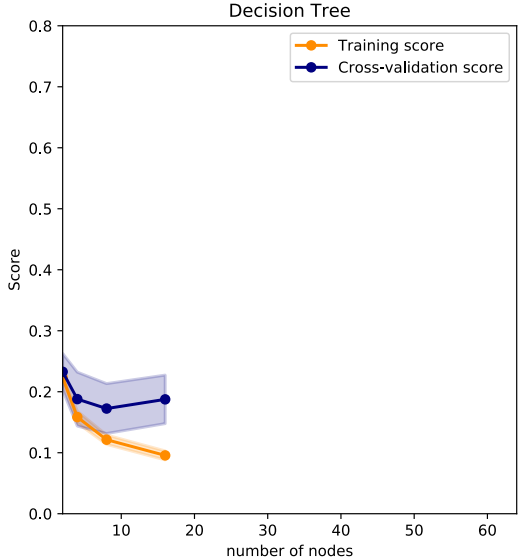


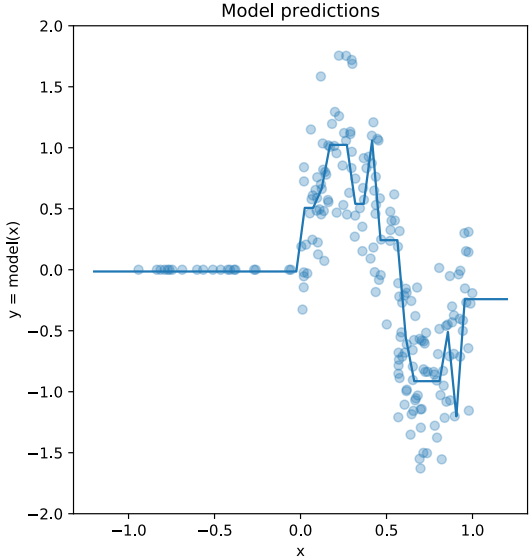
Single Decision Tree



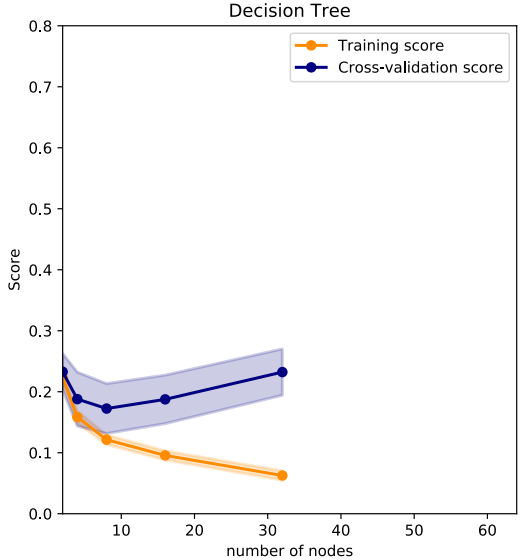


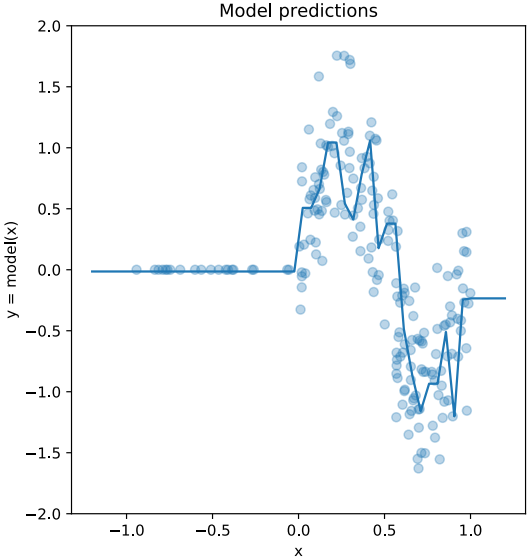
Single Decision Tree



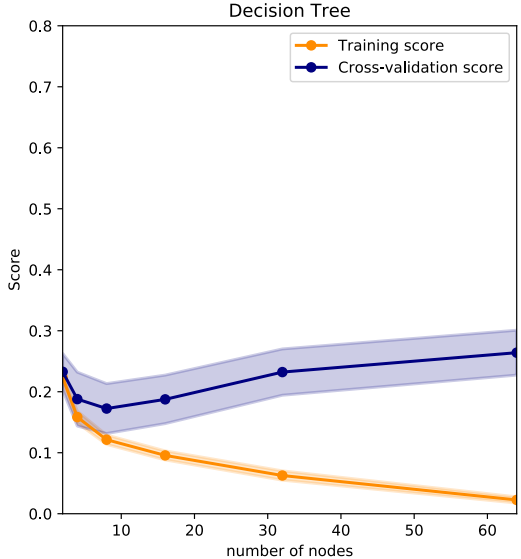


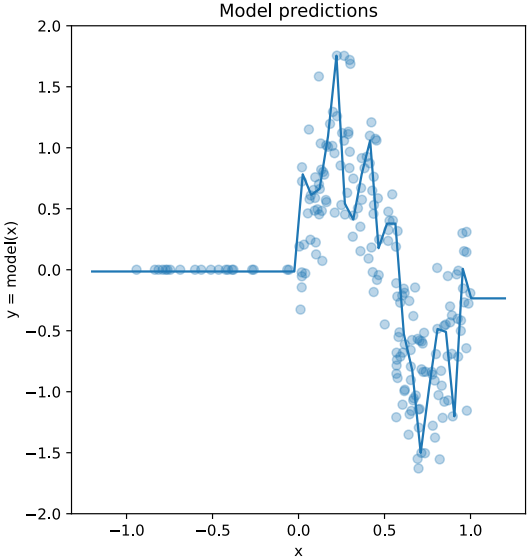
Single Decision Tree



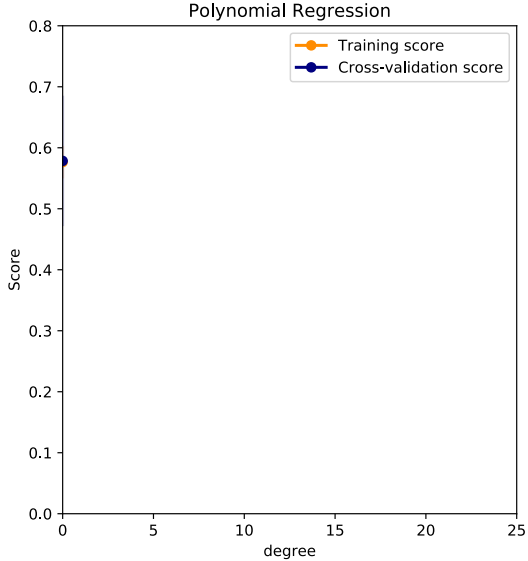


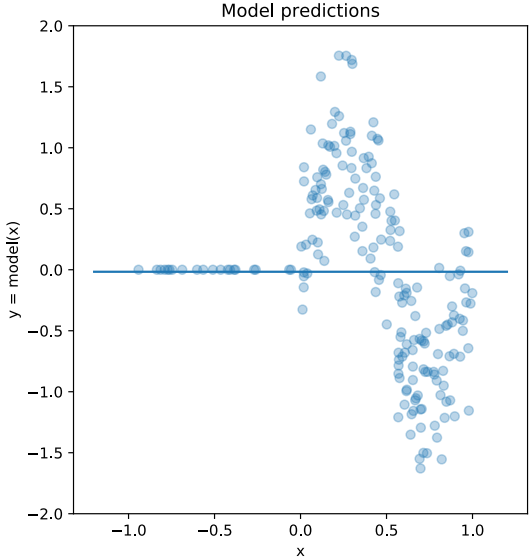
Single Decision Tree



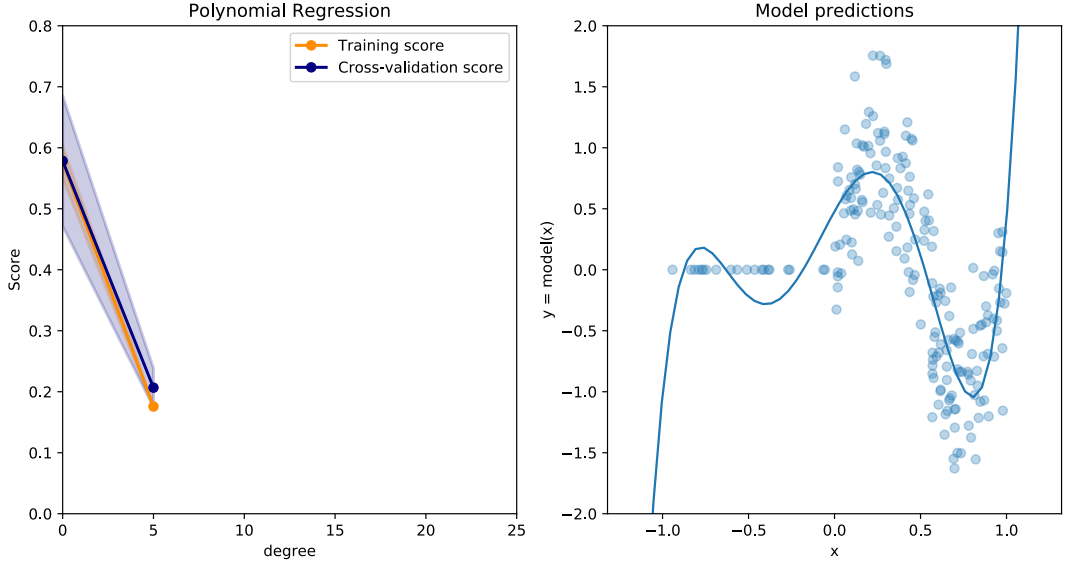


Polynomial regression

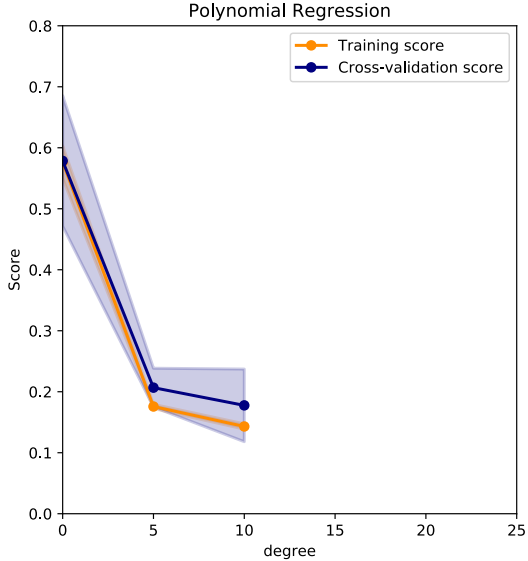


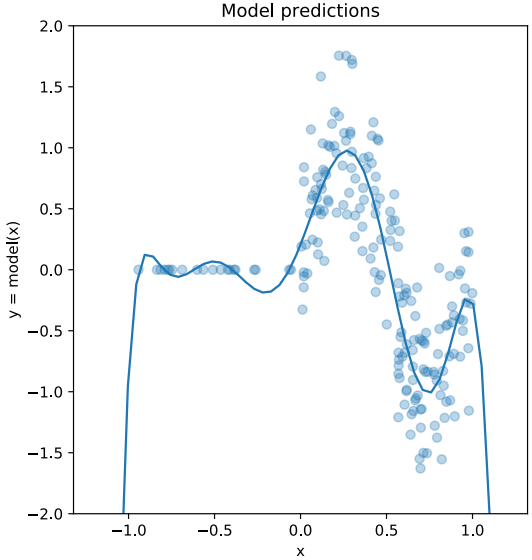


Polynomial regression

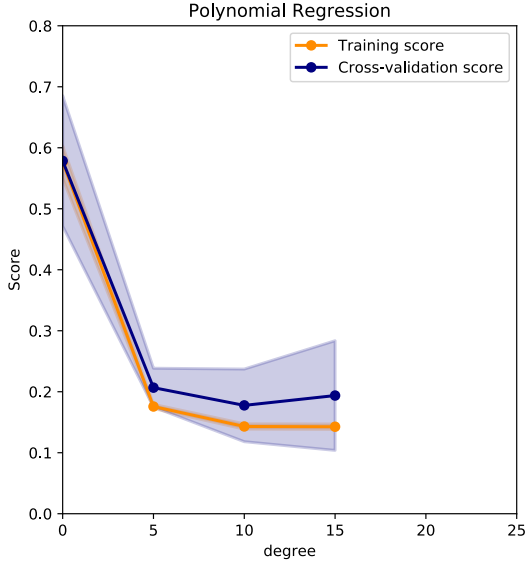


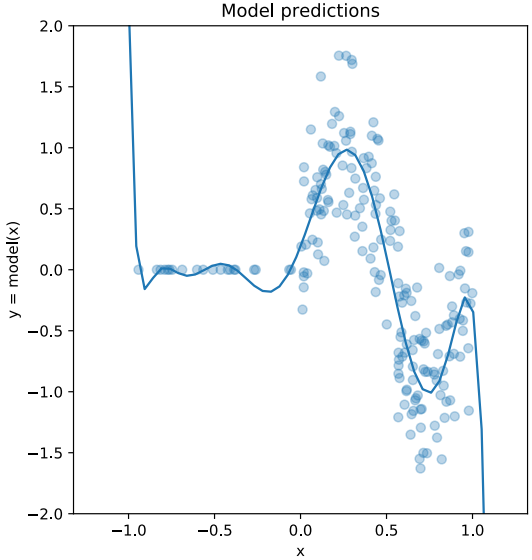
Polynomial regression



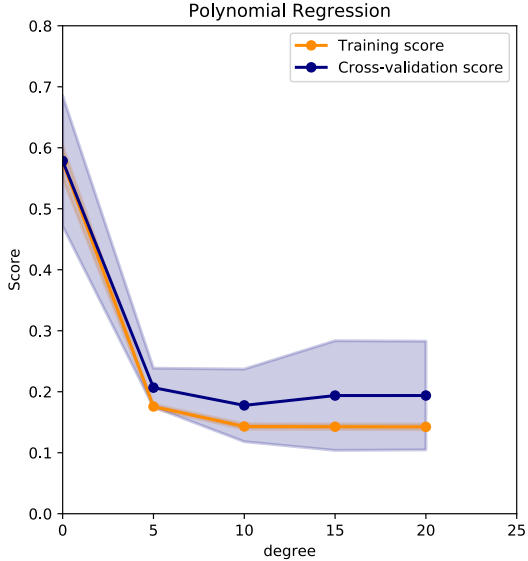


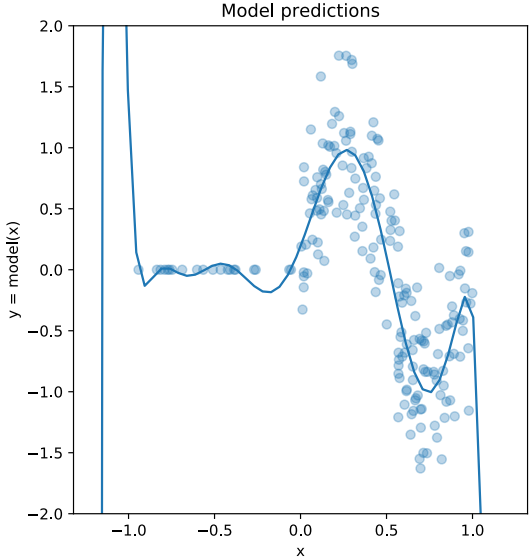
Polynomial regression



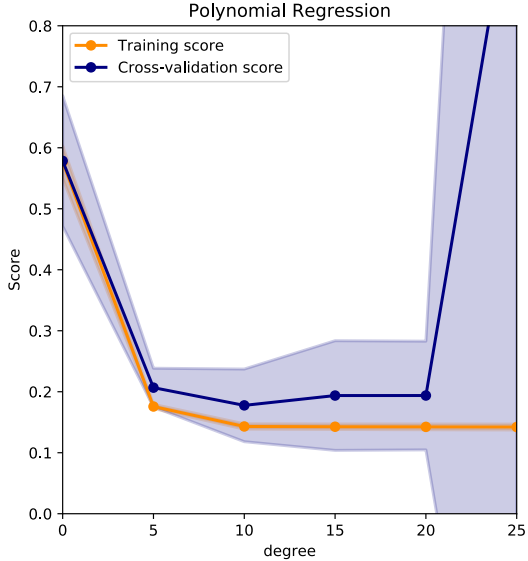


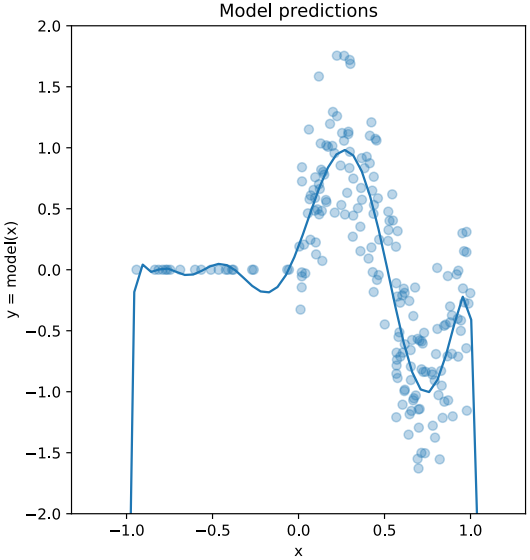
Polynomial regression



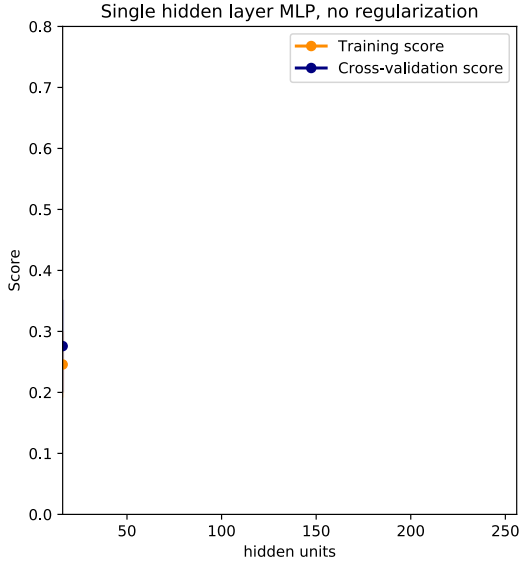


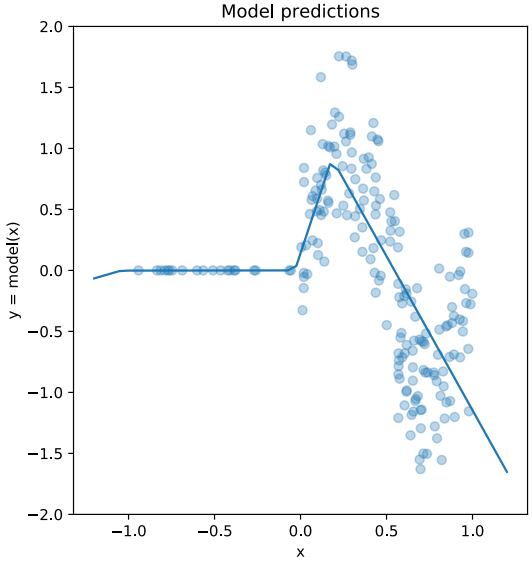
Polynomial regression



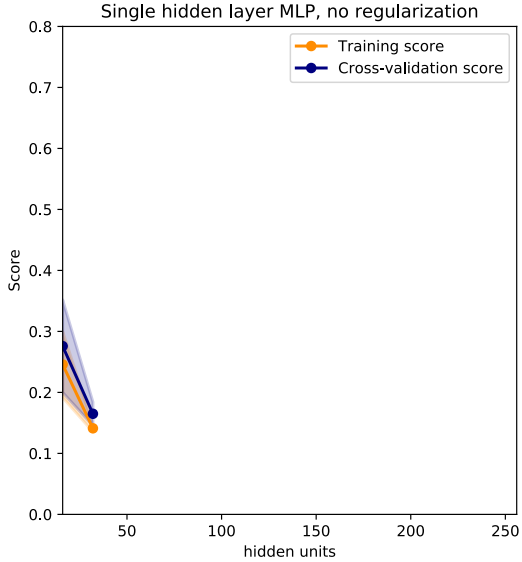


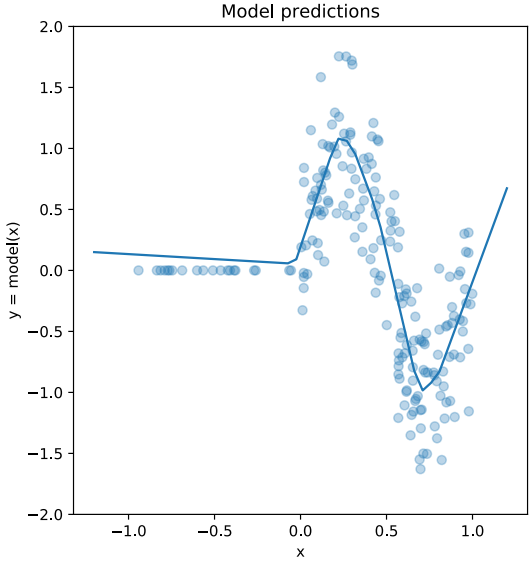
1-hidden layer MLP, no regularization



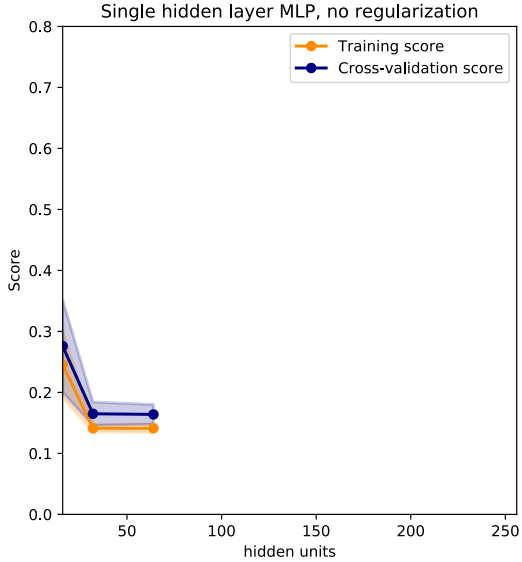


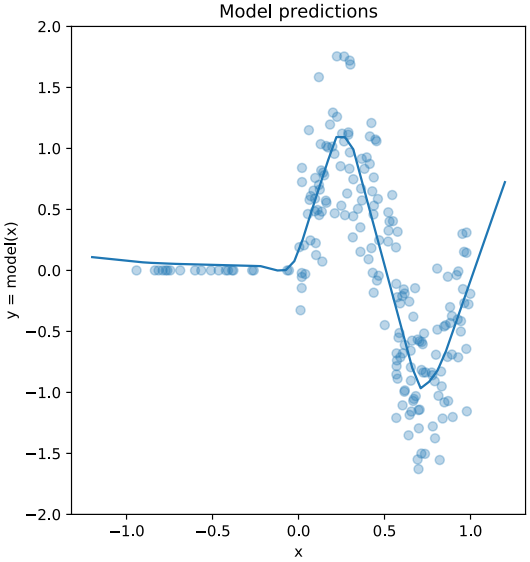
1-hidden layer MLP, no regularization



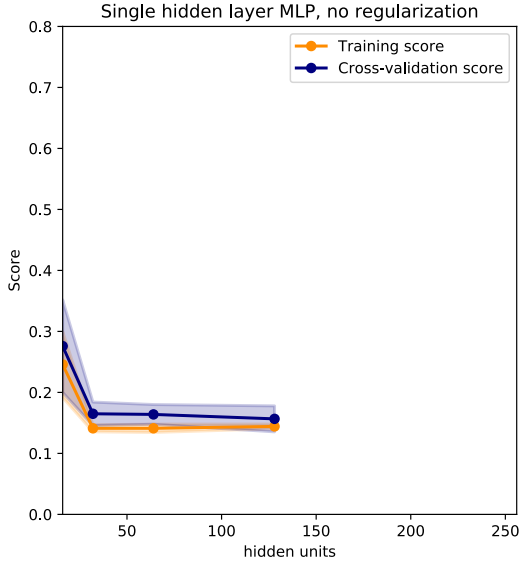


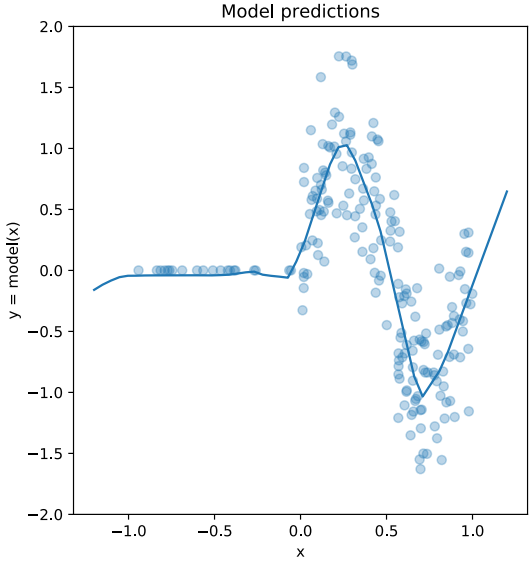
1-hidden layer MLP, no regularization



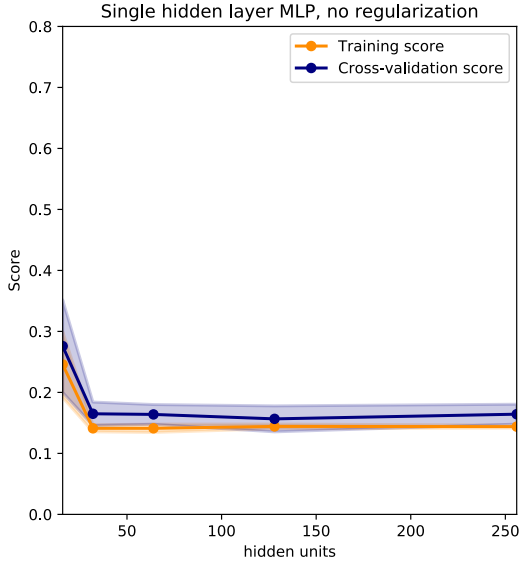


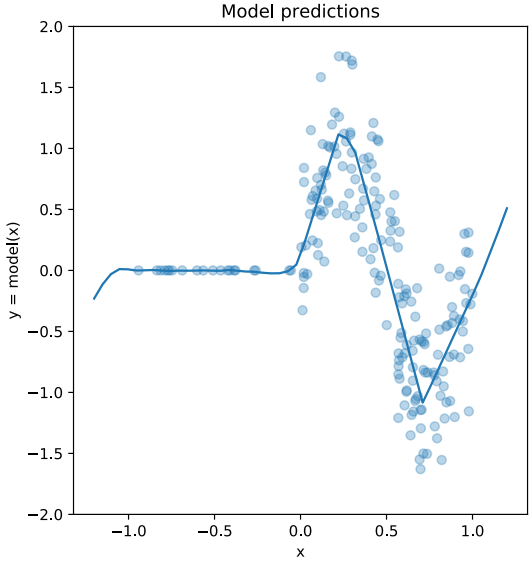
1-hidden layer MLP, no regularization



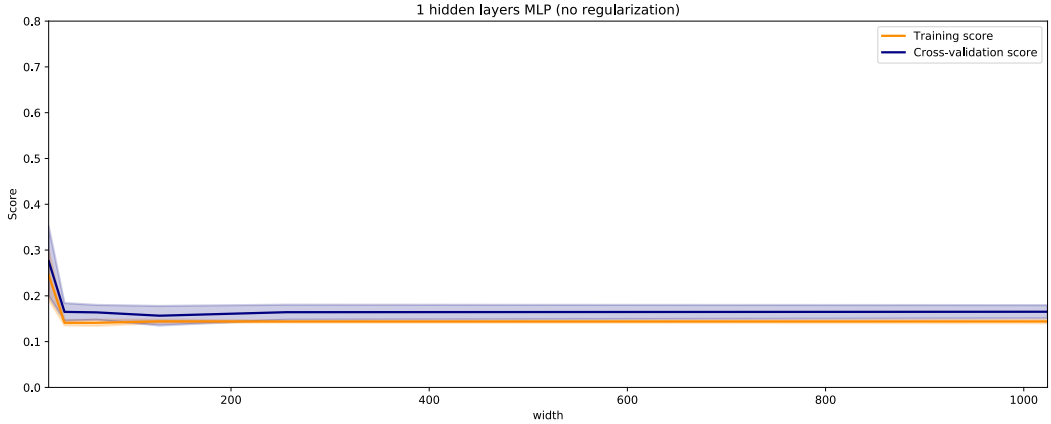


1-hidden layer MLP, no regularization





1-hidden layer MLP, no regularization



Non-intuitive behavior of MLP

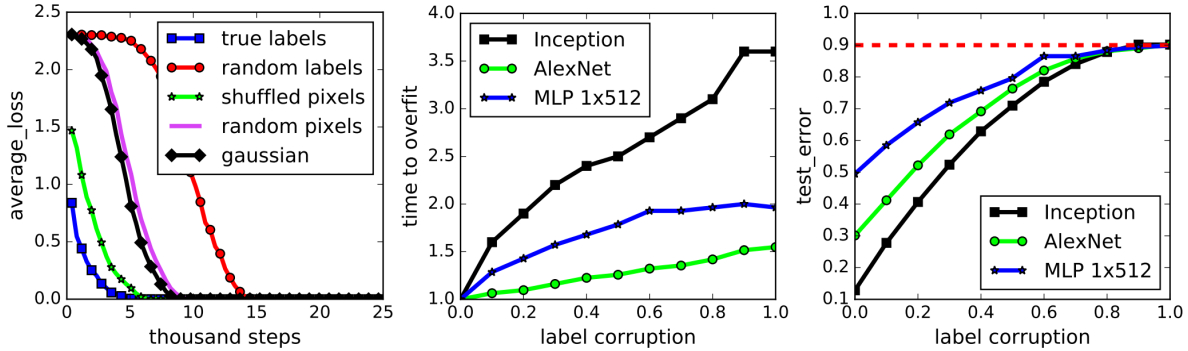
Adding hidden units:

- does not lead to (more) overfitting
- make optimization easier (less steps)
- make computation slower (more FLOPS per pass)

Adding layers:

- does not cause much more overfitting either

Deep Nets Generalize Better than They Should



(a) learning curves (b) convergence slowdown (c) generalization error growth

Figure 1: Fitting random labels and random pixels on CIFAR10. (a) shows the training loss of various experiment settings decaying with the training steps. (b) shows the relative convergence time with different label corruption ratio. (c) shows the test error (also the generalization error since training error is 0) under different label corruptions.

Understanding deep learning requires rethinking generalization, C. Zhang et al., 2016.

The Mistery of Generalization

What makes deep network generalize so well?

- Intrinsic to the piecewise linear nature of the ReLU NN?

The Mistery of Generalization

What makes deep network generalize so well?

- Intrinsic to the piecewise linear nature of the ReLU NN?
- Deep compositional architectures = strong inductive prior?

The Mistery of Generalization

What makes deep network generalize so well?

- Intrinsic to the piecewise linear nature of the ReLU NN?
- Deep compositional architectures = strong inductive prior?
- SGD with large step sizes, small minibatches and early stopping?

The Mistery of Generalization

What makes deep network generalize so well?

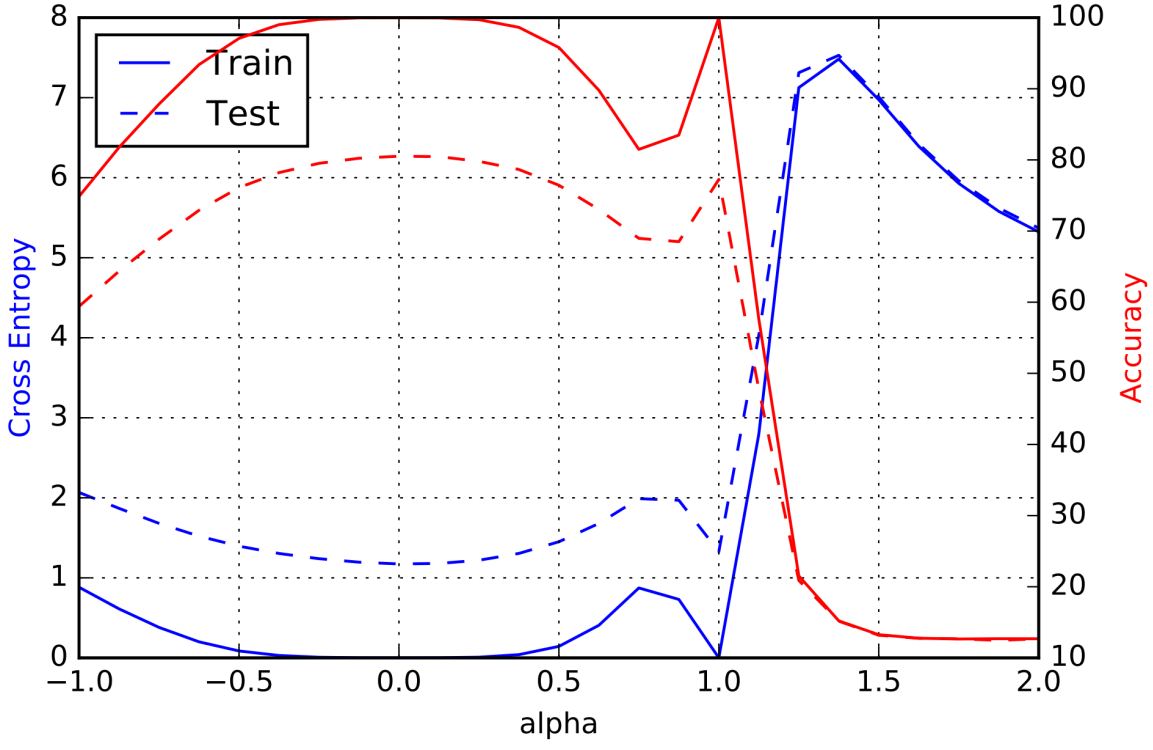
- Intrinsic to the piecewise linear nature of the ReLU NN?
- Deep compositional architectures = strong inductive prior?
- SGD with large step sizes, small minibatches and early stopping?
- Gradient Descent on over-parametrized net from small random init?

The Mistery of Generalization

What makes deep network generalize so well?

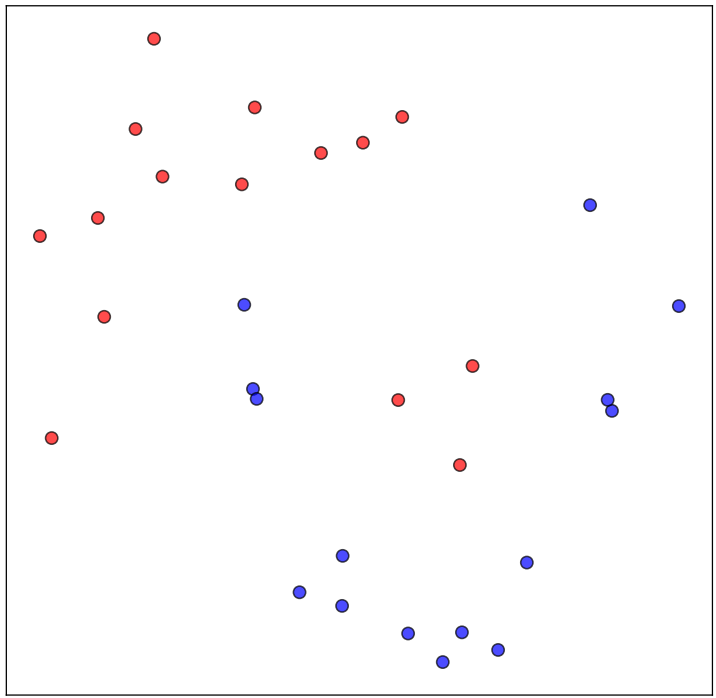
- Intrinsic to the piecewise linear nature of the ReLU NN?
- Deep compositional architectures = strong inductive prior?
- SGD with large step sizes, small minibatches and early stopping?
- Gradient Descent on over-parametrized net from small random init?
- A combination of all of the above?

Small Batch vs Large Batch

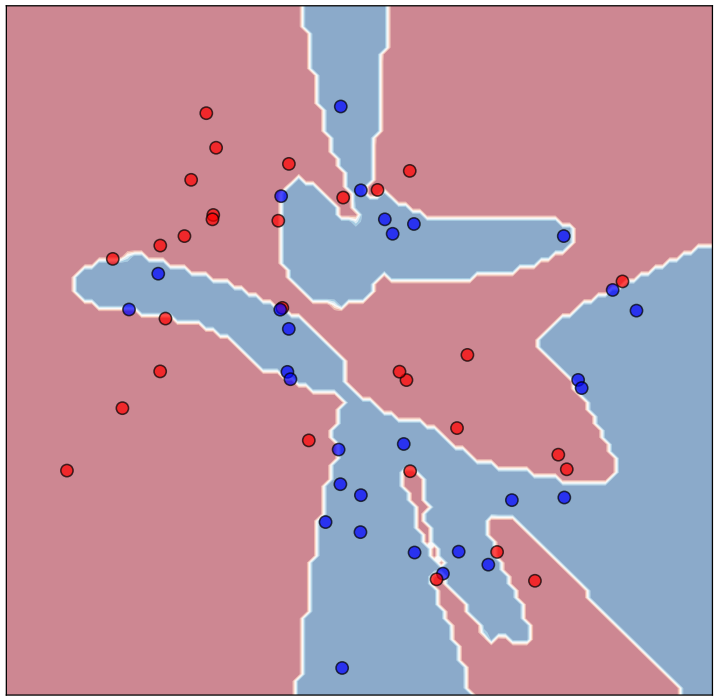


On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima, N. S. Keskar et al., 2016

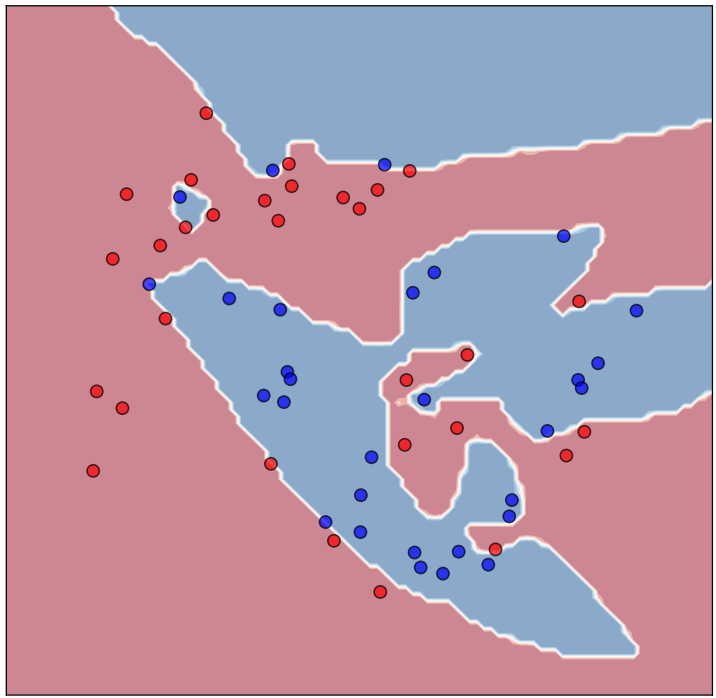
Good and bad global minima



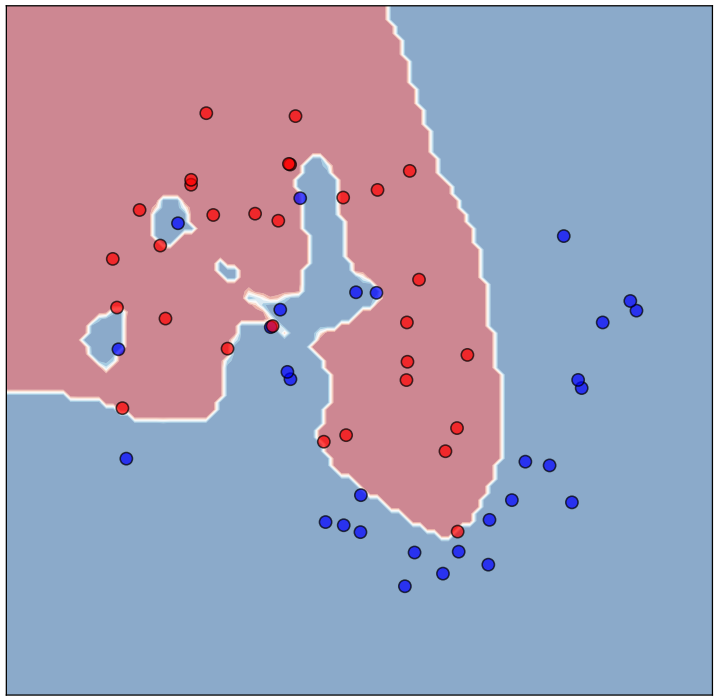
Good and bad global minima



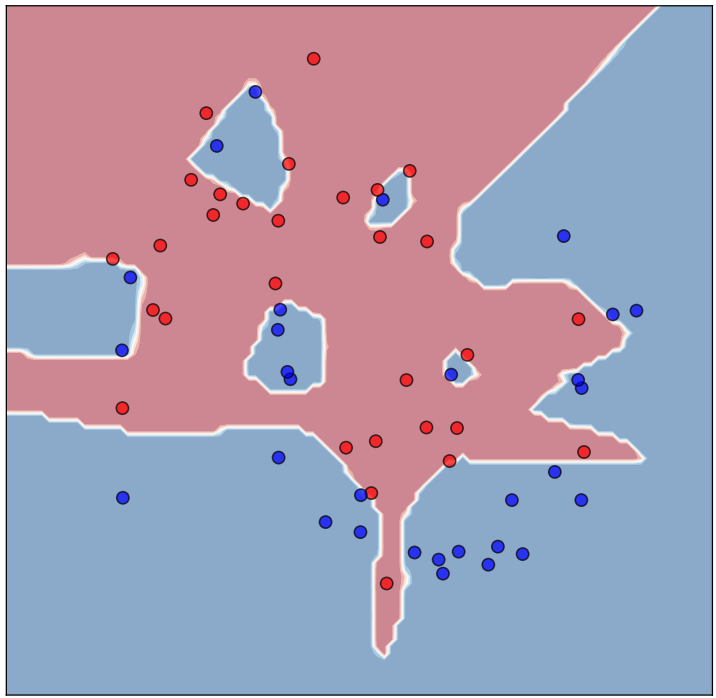
Good and bad global minima



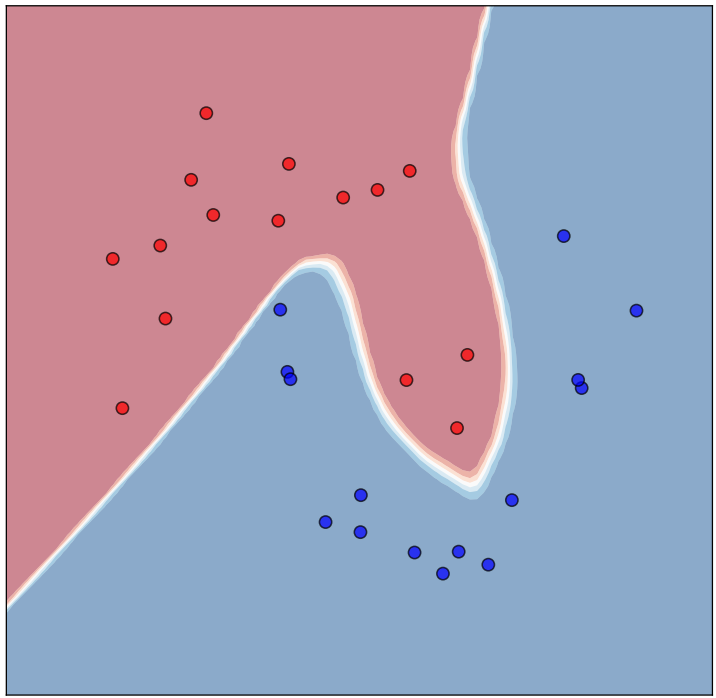
Good and bad global minima



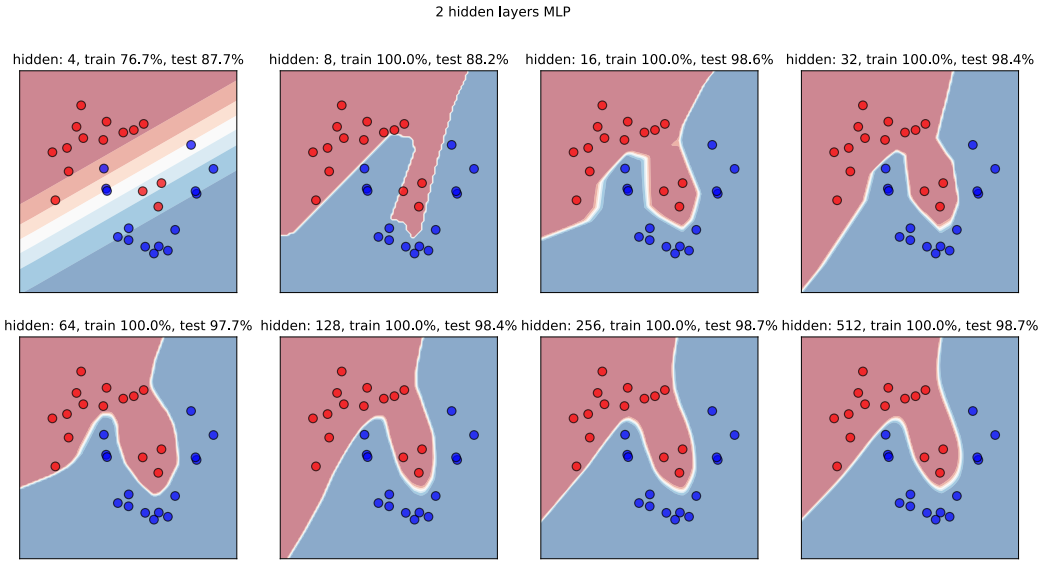
Good and bad global minima



Good and bad global minima



Good and bad global minima



Good and bad global minima

- Typical model capacity is enough to represent bad global minima.

Good and bad global minima

- Typical model capacity is enough to represent bad global minima.
- But even with large batches, GD avoids those pathological global minima.

Good and bad global minima

- Typical model capacity is enough to represent bad global minima.
- But even with large batches, GD avoids those pathological global minima.
- Small-batch / large LR models generalize a bit better but not the main effect.

Good and bad global minima

- Typical model capacity is enough to represent bad global minima.
- But even with large batches, GD avoids those pathological global minima.
- Small-batch / large LR models generalize a bit better but not the main effect.
- Good global minimizers have a wide attractor basin for GD

Towards Understanding Generalization of Deep Learning: Perspective of Loss Landscapes

Lei Wu, Zhanxing Zhu, Weinan E

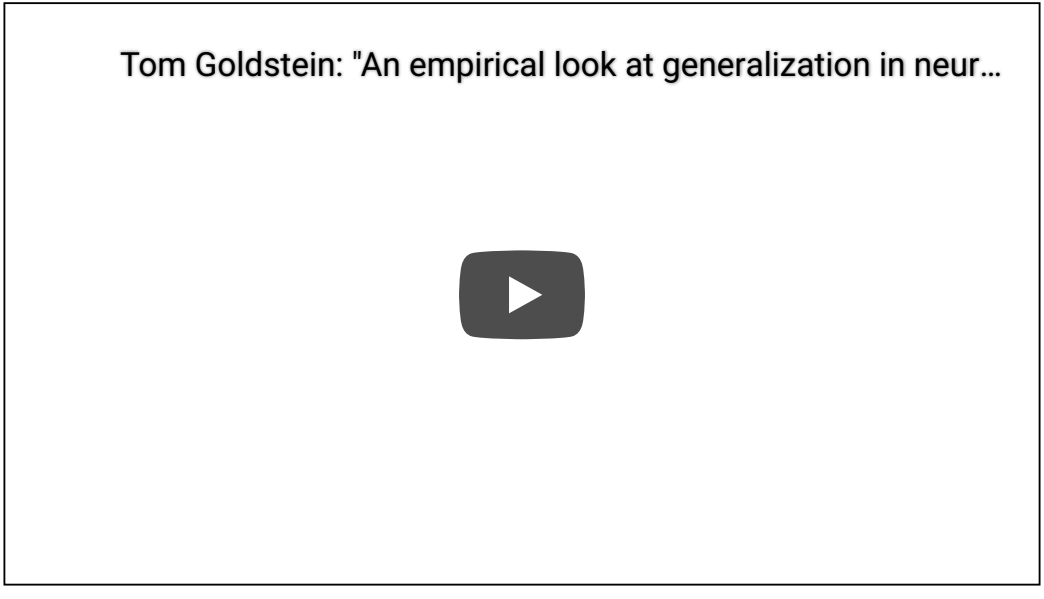
Good and bad global minima

- Typical model capacity is enough to represent bad global minima.
- But even with large batches, GD avoids those pathological global minima.
- Small-batch / large LR models generalize a bit better but not the main effect.
- Good global minimizers have a wide attractor basin for GD

Towards Understanding Generalization of Deep Learning: Perspective of Loss Landscapes

Lei Wu, Zhanxing Zhu, Weinan E

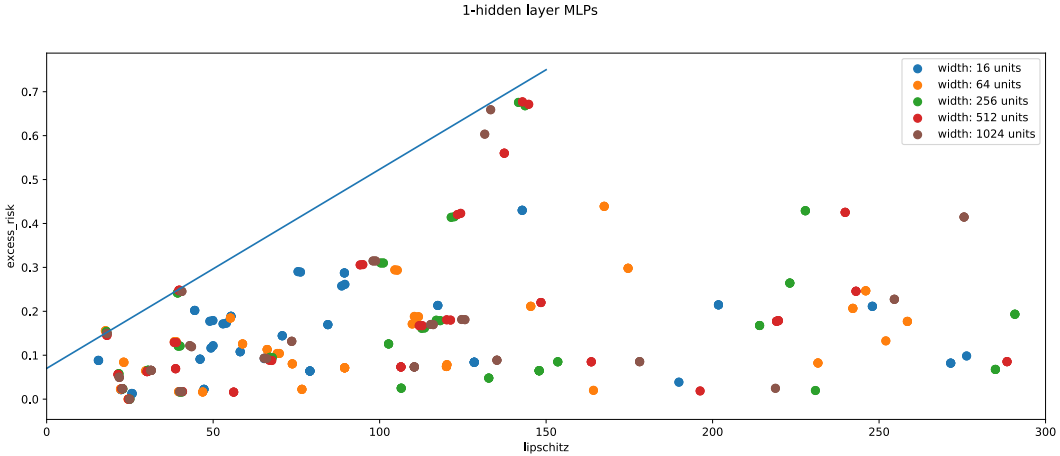
Geometry of the loss



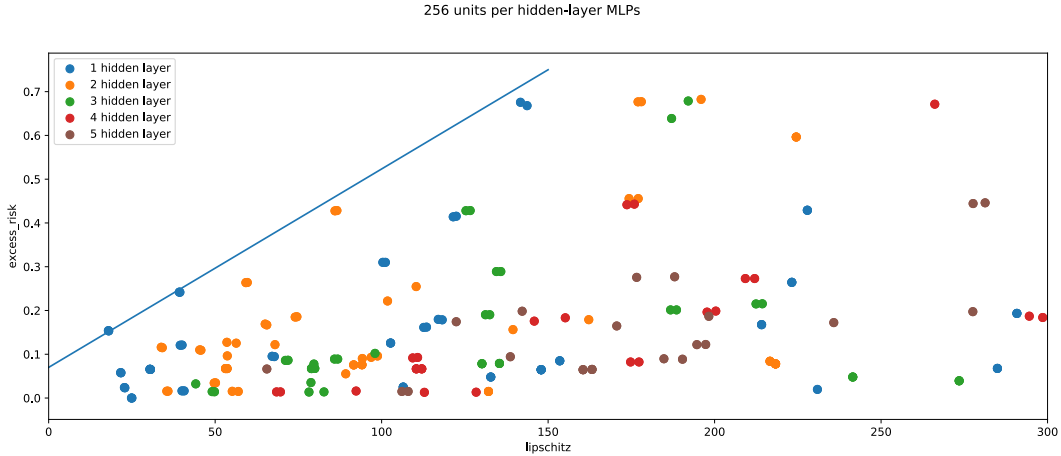
Visualizing the Loss Landscape of Neural Nets Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, Tom Goldstein

Understanding Generalization through Visualizations W. Ronny Huang, Zeyad Emam, Micah Goldblum, Liam Fowl, Justin K. Terry, Furong

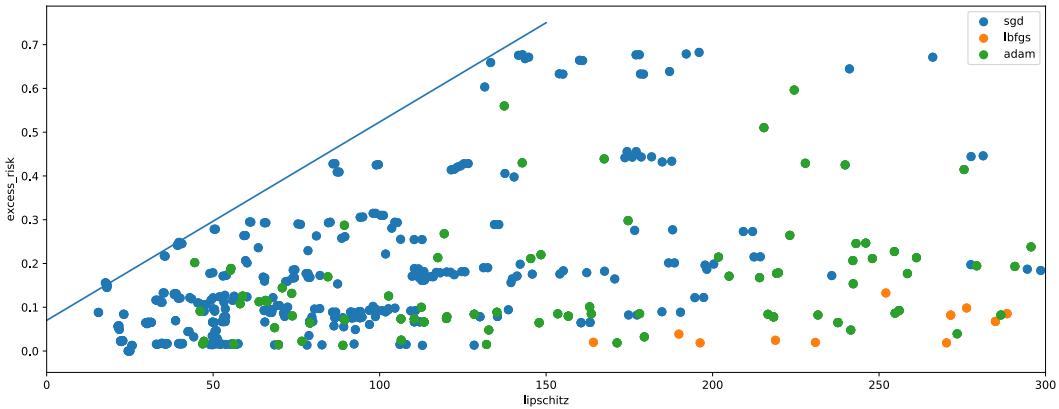
Lipschitz constant as a measure of capacity:



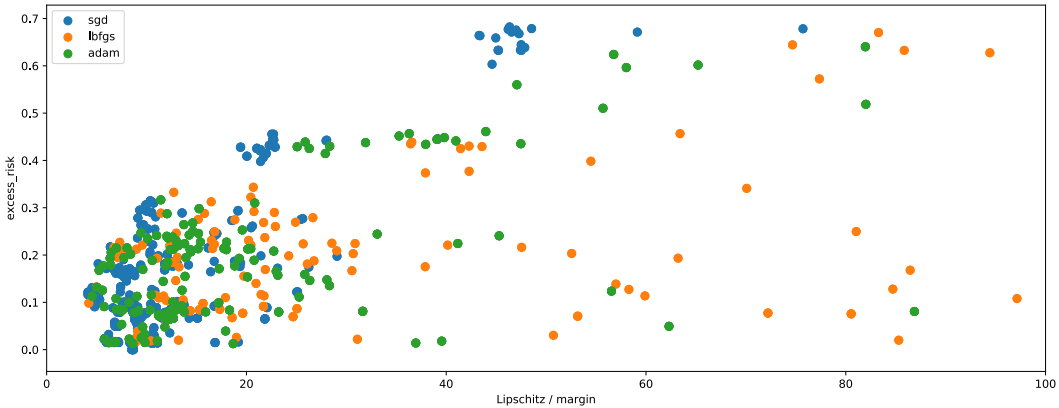
Lipschitz constant as a measure of capacity:



Lipschitz constant as a measure of capacity:



Lipschitz constant as a measure of capacity:



Generalization bounds

Theorem 1.1. *Let nonlinearities $(\sigma_1, \ldots, \sigma_L)$ and reference matrices $(M_1, \ldots, M_L)$ be given as above (i.e., σ_i is ρ_i-Lipschitz and $\sigma_i(0) = 0$). Then for $(x, y), (x_1, y_1), \ldots, (x_n, y_n)$ drawn iid from any probability distribution over $\mathbb{R}^d \times \{1, \ldots, k\}$, with probability at least $1 - \delta$ over $((x_i, y_i))_{i=1}^n$, every margin $\gamma > 0$ and network $F_{\mathcal{A}} : \mathbb{R}^d \to \mathbb{R}^k$ with weight matrices $\mathcal{A} = (A_1, \ldots, A_L)$ satisfy*

$$\Pr\left[\arg\max_j F_{\mathcal{A}}(x)_j \neq y\right] \leq \widehat{\mathcal{R}}_\gamma(F_{\mathcal{A}}) + \widetilde{\mathcal{O}}\left(\frac{\|X\|_2 R_{\mathcal{A}}}{\gamma n}\ln(W) + \sqrt{\frac{\ln(1/\delta)}{n}}\right),$$

where $\widehat{\mathcal{R}}_\gamma(f) \leq n^{-1}\sum_i \mathbb{1}\left[f(x_i)_{y_i} \leq \gamma + \max_{j \neq y_i} f(x_i)_j\right]$ *and* $\|X\|_2 = \sqrt{\sum_i \|x_i\|_2^2}$.

Spectral complexity:

$$R_{\mathcal{A}} := \left(\prod_{i=1}^{L} \rho_i \|A_i\|_\sigma\right)\left(\sum_{i=1}^{L} \frac{\|A_i^\top - M_i^\top\|_{2,1}^{2/3}}{\|A_i\|_\sigma^{2/3}}\right)^{3/2}$$

Spectrally-normalized margin bounds for neural networks Peter Bartlett, Dylan J. Foster, Matus Telgarsky

Generalization bounds

Theorem 4.1. *For any fully connected network f_A with $\rho_\delta \geq 3d$, any probability $0 < \delta \leq 1$ and any margin γ, Algorithm 1 generates weights $\tilde{A}$ for the network $f_{\tilde{A}}$ such that with probability $1 - \delta$ over the training set and $f_{\tilde{A}}$, the expected error $L_0(f_{\tilde{A}})$ is bounded by*

$$\hat{L}_\gamma(f_A) + \tilde{O}\left(\sqrt{\frac{c^2 d^2 \max_{x \in S} \|f_A(x)\|_2^2 \sum_{i=1}^{d} \frac{1}{\mu_i^2 \mu_{i\rightarrow}^2}}{\gamma^2 m}}\right)$$

where μ_i, $\mu_{i\rightarrow}$, c and ρ_δ are layer cushion, interlayer cushion, activation contraction and interlayer smoothness defined in Definitions 4, 5, 6 and 7 respectively.

- Compressible Neural Networks generalize better;
- Compressibility stems from noise stability properties of deep nets.

Stronger generalization bounds for deep nets via a compression approach Sanjeev Arora, Rong Ge, Behnam Neyshabur, Yi Zhang

Conclusion

Conclusions

DL optimization is non-convex but bad local minima and saddle structures are rarely a problem (on common DL tasks).

A stronger optimizer is not necessarily a stronger learner.

Neural Networks are over-parametrized but can still generalize.

(Stochastic) Gradient Descent has a built-in implicit regularizer.

Compressible / low Lipschitz networks generalize better.

Variance in gradient can help with generalization but can hurt final convergence.

We need more theory to guide the design of architectures and optimizers that make **learning** faster with fewer labels.

Practical Takeaways

Overparametrize deep architectures

Design architectures to limit conditioning issues:

- Use skip connections & normalization layers

Use stochastic optimizers that are robust to bad conditioning

Use small minibatches and learge learning rates

Use validation set to anneal learning rate and do early stopping

Is it very often possible to trade more compute for less overfitting with **data augmentation** and **stochastic regularizers** (e.g. dropout).

Lab #8: back here in 15 min!