Approximate Functions

Going Deep

Jonathon Hare

Vision, Learning and Control
University of Southampton

Overview

- No free lunch and universal approximation
- Why go deep?
- Problems of going deep
- Some fixes:
  - Improving gradient flow with skip connections
  - Regularising with Dropout

No Free Lunch

- Statistical learning theory claims that a machine can generalise well from a finite training set.

[1] or perhaps more generally rules which are not certain

No Free Lunch

- Statistical learning theory claims that a machine can generalise well from a finite training set.
- This contradicts basic inductive reasoning which says to derive a rule describing every member of a set one must have information about every member.

[1] or perhaps more generally rules which are not certain

No Free Lunch

- Statistical learning theory claims that a machine can generalise well from a finite training set.
- This contradicts basic inductive reasoning which says to derive a rule describing every member of a set one must have information about every member.
- Machine learning avoids this problem by learning probabilistic[1] rules which are *probably* correct about *most* members of the set they concern.

[1] or perhaps more generally rules which are not certain

No Free Lunch

- Statistical learning theory claims that a machine can generalise well from a finite training set.
- This contradicts basic inductive reasoning which says to derive a rule describing every member of a set one must have information about every member.
- Machine learning avoids this problem by learning probabilistic[1] rules which are *probably* correct about *most* members of the set they concern.
- But, **no free lunch theorem** states that every possible classification machine has the *same error* when averaged over *all possible* data-generating distributions.

[1] or perhaps more generally rules which are not certain

No Free Lunch

- Statistical learning theory claims that a machine can generalise well from a finite training set.
- This contradicts basic inductive reasoning which says to derive a rule describing every member of a set one must have information about every member.
- Machine learning avoids this problem by learning probabilistic[1] rules which are *probably* correct about *most* members of the set they concern.
- But, **no free lunch theorem** states that every possible classification machine has the *same error* when averaged over *all possible* data-generating distributions.
  - **No machine learning algorithm is universally better than any other!**

[1] or perhaps more generally rules which are not certain

No Free Lunch

- Statistical learning theory claims that a machine can generalise well from a finite training set.
- This contradicts basic inductive reasoning which says to derive a rule describing every member of a set one must have information about every member.
- Machine learning avoids this problem by learning probabilistic[1] rules which are *probably* correct about *most* members of the set they concern.
- But, **no free lunch theorem** states that every possible classification machine has the *same error* when averaged over *all possible* data-generating distributions.
  - **No machine learning algorithm is universally better than any other!**
  - Fortunately, in the real world, data is generated by a small subset of generating distributions...

[1] or perhaps more generally rules which are not certain

The Universal Approximation Theorem

Let $\psi : \mathbb{R} \to \mathbb{R}$ be a nonconstant, bounded, and continuous function.

The Universal Approximation Theorem

Let $\psi : \mathbb{R} \to \mathbb{R}$ be a nonconstant, bounded, and continuous function. Let I_m denote the m-dimensional unit hypercube $[0, 1]^m$.

The Universal Approximation Theorem

Let $\psi : \mathbb{R} \rightarrow \mathbb{R}$ be a nonconstant, bounded, and continuous function. Let I_m denote the m-dimensional unit hypercube $[0, 1]^m$. The space of real-valued continuous functions on I_m is denoted by $C(I_m)$.

The Universal Approximation Theorem

Let $\psi : \mathbb{R} \rightarrow \mathbb{R}$ be a nonconstant, bounded, and continuous function. Let I_m denote the m-dimensional unit hypercube $[0, 1]^m$. The space of real-valued continuous functions on I_m is denoted by $C(I_m)$. Then, given any $\varepsilon > 0$ and any function $f \in C(I_m)$, there exist an integer N, real constants $v_i, b_i \in \mathbb{R}$ and real vectors $w_i \in \mathbb{R}^m$ for $i = 1, \dots, N$, such that we may define:

$F(x) = \sum_{i=1}^{N} v_i \psi({w_i}^T x + b_i)$ as an approximate realization of the function f ; that is,

The Universal Approximation Theorem

Let $\psi : \mathbb{R} \to \mathbb{R}$ be a nonconstant, bounded, and continuous function. Let I_m denote the m-dimensional unit hypercube $[0, 1]^m$. The space of real-valued continuous functions on I_m is denoted by $C(I_m)$. Then, given any $\varepsilon > 0$ and any function $f \in C(I_m)$, there exist an integer N, real constants $v_i, b_i \in \mathbb{R}$ and real vectors $w_i \in \mathbb{R}^m$ for $i = 1, \dots, N$, such that we may define:

$F(x) = \sum_{i=1}^{N} v_i \psi(w_i^T x + b_i)$ as an approximate realization of the function f ; that is,

$|F(x) - f(x)| < \varepsilon \; \forall \; x \in I_m$.

The Universal Approximation Theorem

Let $\psi : \mathbb{R} \to \mathbb{R}$ be a nonconstant, bounded, and continuous function. Let I_m denote the m-dimensional unit hypercube $[0,1]^m$. The space of real-valued continuous functions on I_m is denoted by $C(I_m)$. Then, given any $\varepsilon > 0$ and any function $f \in C(I_m)$, there exist an integer N, real constants $v_i, b_i \in \mathbb{R}$ and real vectors $w_i \in \mathbb{R}^m$ for $i = 1, \ldots, N$, such that we may define:

$F(x) = \sum_{i=1}^{N} v_i \psi(w_i^T x + b_i)$ as an approximate realization of the function f ; that is,

$|F(x) - f(x)| < \varepsilon \;\forall\; x \in I_m$.

$\implies$ simple neural networks can represent a wide variety of interesting functions when given appropriate parameters.

So a single hidden layer network can approximate most functions?

- Yes!

So a single hidden layer network can approximate most functions?

- Yes!
- But, ...

So a single hidden layer network can approximate most functions?

- Yes!
- But, ...
  - to get the precision you require (small ε), you might need a really large number of hidden units (very large N).

So a single hidden layer network can approximate most functions?

- Yes!
- But, ...
  - to get the precision you require (small ε), you might need a really large number of hidden units (very large N).
  - worse-case analysis shows it might be exponential (possibly one hidden unit for *every* input configuration)

So a single hidden layer network can approximate most functions?

- Yes!
- But, ...
  - to get the precision you require (small ε), you might need a really large number of hidden units (very large N).
  - worse-case analysis shows it might be exponential (possibly one hidden unit for *every* input configuration)
  - We've not said anything about learnability...

So a single hidden layer network can approximate most functions?

- Yes!
- But, ...
  - to get the precision you require (small ε), you might need a really large number of hidden units (very large N).
  - worse-case analysis shows it might be exponential (possibly one hidden unit for *every* input configuration)
  - We've not said anything about learnability...
    - The optimiser might not find a good solution[2].

[2] note that it has been shown that the gradients of the function are approximated by the network to an arbitrary precision

So a single hidden layer network can approximate most functions?

- Yes!
- But, ...
  - to get the precision you require (small ε), you might need a really large number of hidden units (very large N).
  - worse-case analysis shows it might be exponential (possibly one hidden unit for *every* input configuration)
  - We've not said anything about learnability...
    - The optimiser might not find a good solution[2].
    - The training algorithm might just choose the wrong solution as a result of overfitting.

[2] note that it has been shown that the gradients of the function are approximated by the network to an arbitrary precision

So a single hidden layer network can approximate most functions?

- Yes!
- But, ...
  - to get the precision you require (small ε), you might need a really large number of hidden units (very large N).
  - worse-case analysis shows it might be exponential (possibly one hidden unit for *every* input configuration)
  - We've not said anything about learnability...
    - The optimiser might not find a good solution[2].
    - The training algorithm might just choose the wrong solution as a result of overfitting.
    - *There is no known universal procedure for examining a set of examples and choosing a function that will generalise to points out of the training set.*

[2] note that it has been shown that the gradients of the function are approximated by the network to an arbitrary precision

Then Why Go Deep?

- There are functions you can compute with a deep neural network that shallow networks require exponentially more hidden units to compute.
  - The following function is more efficient to implement using a deep neural network: $y = x_1 \oplus x_2 \oplus x_3 \oplus \cdots \oplus x_n$

Then Why Go Deep?

- There are functions you can compute with a deep neural network that shallow networks require exponentially more hidden units to compute.
  - The following function is more efficient to implement using a deep neural network: $y = x_1 \oplus x_2 \oplus x_3 \oplus \cdots \oplus x_n$
- We should care about the data generating distribution (c.f. NFL).

Then Why Go Deep?

- There are functions you can compute with a deep neural network that shallow networks require exponentially more hidden units to compute.
  - The following function is more efficient to implement using a deep neural network: $y = x_1 \oplus x_2 \oplus x_3 \oplus \cdots \oplus x_n$
- We should care about the data generating distribution (c.f. NFL).
  - Real-world data has significant structure; often believed to be generated by (relatively) simple low-dimensional latent processes.

Then Why Go Deep?

- There are functions you can compute with a deep neural network that shallow networks require exponentially more hidden units to compute.
  - The following function is more efficient to implement using a deep neural network: $y = x_1 \oplus x_2 \oplus x_3 \oplus \cdots \oplus x_n$
- We should care about the data generating distribution (c.f. NFL).
  - Real-world data has significant structure; often believed to be generated by (relatively) simple low-dimensional latent processes.
  - Implies a prior belief that the underlying factors of variation in data can be explained by a hierarchical composition of increasingly simple latent factors

Then Why Go Deep?

- There are functions you can compute with a deep neural network that shallow networks require exponentially more hidden units to compute.
  - The following function is more efficient to implement using a deep neural network: $y = x_1 \oplus x_2 \oplus x_3 \oplus \cdots \oplus x_n$
- We should care about the data generating distribution (c.f. NFL).
  - Real-world data has significant structure; often believed to be generated by (relatively) simple low-dimensional latent processes.
  - Implies a prior belief that the underlying factors of variation in data can be explained by a hierarchical composition of increasingly simple latent factors
- Alternatively, one could just consider that a deep architecture just expresses that the function we wish to learn is a program made of multiple steps where each step makes use of the previous steps outputs.

Then Why Go Deep?

- There are functions you can compute with a deep neural network that shallow networks require exponentially more hidden units to compute.
  - The following function is more efficient to implement using a deep neural network: $y = x_1 \oplus x_2 \oplus x_3 \oplus \cdots \oplus x_n$
- We should care about the data generating distribution (c.f. NFL).
  - Real-world data has significant structure; often believed to be generated by (relatively) simple low-dimensional latent processes.
  - Implies a prior belief that the underlying factors of variation in data can be explained by a hierarchical composition of increasingly simple latent factors
- Alternatively, one could just consider that a deep architecture just expresses that the function we wish to learn is a program made of multiple steps where each step makes use of the previous steps outputs.
- **Empirically, deeper networks just seem to generalise better!**

What are the problems?

- Learnability is still hard

What are the problems?

- Learnability is still hard
  - Problems of gradient flow

What are the problems?

- Learnability is still hard
  - Problems of gradient flow
  - Horrible symmetries in the loss landscape

What are the problems?

- Learnability is still hard
  - Problems of gradient flow
  - Horrible symmetries in the loss landscape
  - Overfitting

Vanishing and Exploding Gradients

- The vanishing and exploding gradient problem is a difficulty found in training NN with gradient-based learning methods and backpropagation.

Vanishing and Exploding Gradients

- The vanishing and exploding gradient problem is a difficulty found in training NN with gradient-based learning methods and backpropagation.
- In training, the gradient may become vanishingly small (or large), effectively preventing the weight from changing its value (or exploding in value).

Vanishing and Exploding Gradients

- The vanishing and exploding gradient problem is a difficulty found in training NN with gradient-based learning methods and backpropagation.
- In training, the gradient may become vanishingly small (or large), effectively preventing the weight from changing its value (or exploding in value).
- This leads to the neural network not being able to train.

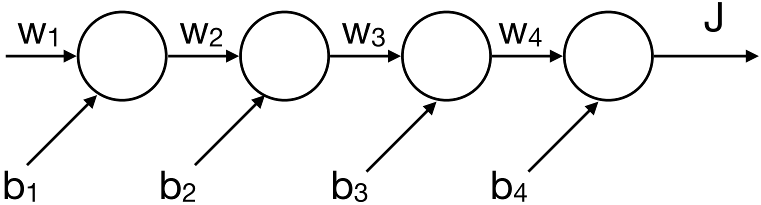
Vanishing and Exploding Gradients

- The vanishing and exploding gradient problem is a difficulty found in training NN with gradient-based learning methods and backpropagation.
- In training, the gradient may become vanishingly small (or large), effectively preventing the weight from changing its value (or exploding in value).
- This leads to the neural network not being able to train.
- This issue affects many-layered networks (feed-forward), as well as recurrent networks.

Vanishing and Exploding Gradients

- The vanishing and exploding gradient problem is a difficulty found in training NN with gradient-based learning methods and backpropagation.
- In training, the gradient may become vanishingly small (or large), effectively preventing the weight from changing its value (or exploding in value).
- This leads to the neural network not being able to train.
- This issue affects many-layered networks (feed-forward), as well as recurrent networks.
- In principle, optimisers that rescale the gradients of each weight should be able to deal with this issue (as long as numeric precision doesn't become problematic).

Issues with Going Deep



Residual Connections

- One of the most effective ways to resolve diminishing gradients is with residual neural networks (ResNets)[3].

[3] K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition," CVPR, Las Vegas, NV, 2016, pp. 770-778.

Residual Connections

- One of the most effective ways to resolve diminishing gradients is with residual neural networks (ResNets)[3].
- ResNets are artificial neural networks that use *skip connections* to jump over layers.

[3] K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition," CVPR, Las Vegas, NV, 2016, pp. 770-778.

Residual Connections

- One of the most effective ways to resolve diminishing gradients is with residual neural networks (ResNets)[3].
- ResNets are artificial neural networks that use *skip connections* to jump over layers.
- The vanishing gradient problem is mitigated in ResNets by reusing activations from a previous layer.

[3] K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition," CVPR, Las Vegas, NV, 2016, pp. 770-778.

Residual Connections

- One of the most effective ways to resolve diminishing gradients is with residual neural networks (ResNets)[3].
- ResNets are artificial neural networks that use *skip connections* to jump over layers.
- The vanishing gradient problem is mitigated in ResNets by reusing activations from a previous layer.
- Is this the full story though? Skip connections also break symmetries, which could be much more important...

[3] K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition," CVPR, Las Vegas, NV, 2016, pp. 770-778.

Residual Connections

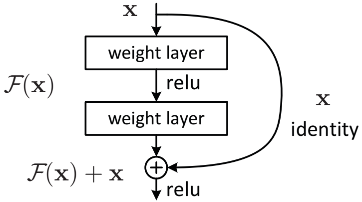


Figure 2. Residual learning: a building block.

K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition," CVPR, Las Vegas, NV, 2016, pp. 770-778.

Residual Connections

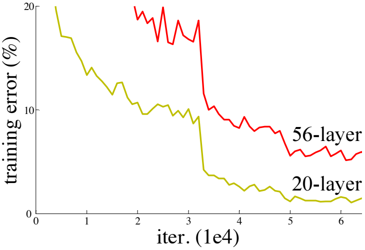


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition," CVPR, Las Vegas, NV, 2016, pp. 770-778.

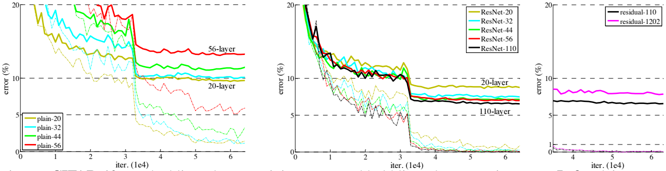


Figure 6. Training on **CIFAR-10**. Dashed lines denote training error, and bold lines denote testing error. **Left**: plain networks. The error of plain-110 is higher than 60% and not displayed. **Middle**: ResNets. **Right**: ResNets with 110 and 1202 layers.

K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition," CVPR, Las Vegas, NV, 2016, pp. 770-778.

Overfitting

- Neural networks with a large number of parameters (and hidden layers) are powerful, however, overfitting is a serious problem in such systems.

Overfitting

- Neural networks with a large number of parameters (and hidden layers) are powerful, however, overfitting is a serious problem in such systems.
- Just as you've seen in simple machines (e.g. Ridge Regression and LASSO), regularisation can help mitigate overfitting

Overfitting

- Neural networks with a large number of parameters (and hidden layers) are powerful, however, overfitting is a serious problem in such systems.
- Just as you've seen in simple machines (e.g. Ridge Regression and LASSO), regularisation can help mitigate overfitting
- In deep networks, we might:

Overfitting

- Neural networks with a large number of parameters (and hidden layers) are powerful, however, overfitting is a serious problem in such systems.
- Just as you've seen in simple machines (e.g. Ridge Regression and LASSO), regularisation can help mitigate overfitting
- In deep networks, we might:
    - Use the architecture to regularise (e.g. ConvNets)

Overfitting

- Neural networks with a large number of parameters (and hidden layers) are powerful, however, overfitting is a serious problem in such systems.
- Just as you've seen in simple machines (e.g. Ridge Regression and LASSO), regularisation can help mitigate overfitting
- In deep networks, we might:
  - Use the architecture to regularise (e.g. ConvNets)
  - Use weight regularisers (L1, L2 [weight decay], etc, ...)

Overfitting

- Neural networks with a large number of parameters (and hidden layers) are powerful, however, overfitting is a serious problem in such systems.
- Just as you've seen in simple machines (e.g. Ridge Regression and LASSO), regularisation can help mitigate overfitting
- In deep networks, we might:
  - Use the architecture to regularise (e.g. ConvNets)
  - Use weight regularisers (L1, L2 [weight decay], etc, ...)
  - Use a stochastic weight regulariser (like dropout)

Overfitting

- Neural networks with a large number of parameters (and hidden layers) are powerful, however, overfitting is a serious problem in such systems.
- Just as you've seen in simple machines (e.g. Ridge Regression and LASSO), regularisation can help mitigate overfitting
- In deep networks, we might:
  - Use the architecture to regularise (e.g. ConvNets)
  - Use weight regularisers (L1, L2 [weight decay], etc, ...)
  - Use a stochastic weight regulariser (like dropout)
  - Regularise by smoothing the optimisation landscape (e.g. Batch Normalisation)

Dropout

- Dropout is a form of regularisation

Dropout

- Dropout is a form of regularisation
- The key idea in dropout is to randomly drop neurons, including all of the connections, from the neural network during training.

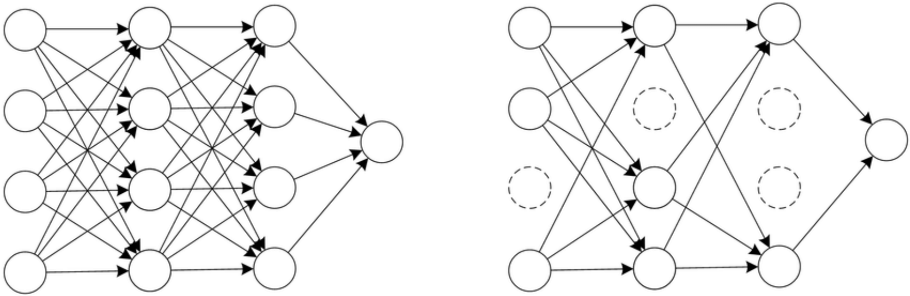
Dropout

- Dropout is a form of regularisation
- The key idea in dropout is to randomly drop neurons, including all of the connections, from the neural network during training.
- Motivation: the best way to regularise a fixed size model is to average predictions over all possible parameter settings, weighting each setting by the posterior probability given the training data.

Dropout

- Dropout is a form of regularisation
- The key idea in dropout is to randomly drop neurons, including all of the connections, from the neural network during training.
- Motivation: the best way to regularise a fixed size model is to average predictions over all possible parameter settings, weighting each setting by the posterior probability given the training data.
  - Clearly this isn't actually tractable - dropout is an approximation of this idea.
  - The idea of averaging predictions to resolve the bias-variance dilemma is called ensembling.

Dropout



(a) Standard Neural Network

(b) Network after Dropout

Image from: `https://www.researchgate.net/figure/Dropout-neural-network-model-a-is-a-standard-neural-network-b-is-the-same-network_fig3_309206911`

How Does Dropout Work?

- In the learning phase, we set a dropout probability for each layer in the network.
- For each batch we then randomly decide whether or not a given neuron in a given layer is removed.

How Does Dropout Work?

- In the learning phase, we set a dropout probability for each layer in the network.
- For each batch we then randomly decide whether or not a given neuron in a given layer is removed.
- **Inverse Dropout** scales the activations with their probability to maintain the overall magnitude of the response when dropout is disabled at evaluation/test time.

How is Inverted Dropout implemented?

- We define a random binary mask $\boldsymbol{m}^{(l)}$ which is used to remove neurons and is generated by sampling a Bernoulli distribution with $P(x = 1) = p$, and note, $m^{(l)}$ changes for each iteration of the backpropagation algorithm.

How is Inverted Dropout implemented?

- We define a random binary mask $\boldsymbol{m}^{(l)}$ which is used to remove neurons and is generated by sampling a Bernoulli distribution with $P(x=1)=p$, and note, $m^{(l)}$ changes for each iteration of the backpropagation algorithm.
- The forward pass of a Dropout layer (function) during **training** is given by $f(\boldsymbol{x}) = \boldsymbol{x} \odot \boldsymbol{m}/p$.
- The forward pass of a Dropout layer (function) during **inference** is given by $f(\boldsymbol{x}) = \boldsymbol{x}$.

How is Inverted Dropout implemented?

- We define a random binary mask $\boldsymbol{m}^{(l)}$ which is used to remove neurons and is generated by sampling a Bernoulli distribution with $P(x = 1) = p$, and note, $m^{(l)}$ changes for each iteration of the backpropagation algorithm.
- The forward pass of a Dropout layer (function) during **training** is given by $f(\boldsymbol{x}) = \boldsymbol{x} \odot \boldsymbol{m}/p$.
- The forward pass of a Dropout layer (function) during **inference** is given by $f(\boldsymbol{x}) = \boldsymbol{x}$.
- This can be applied to any layer(s) of the network *except the output layer!*

How is Inverted Dropout implemented?

- We define a random binary mask $\boldsymbol{m}^{(l)}$ which is used to remove neurons and is generated by sampling a Bernoulli distribution with $P(x = 1) = p$, and note, $m^{(l)}$ changes for each iteration of the backpropagation algorithm.
- The forward pass of a Dropout layer (function) during **training** is given by $f(\boldsymbol{x}) = \boldsymbol{x} \odot \boldsymbol{m}/p$.
- The forward pass of a Dropout layer (function) during **inference** is given by $f(\boldsymbol{x}) = \boldsymbol{x}$.
- This can be applied to any layer(s) of the network *except the output layer!*
- It's not common to put it everywhere; just a couple of select places (empirically chosen).

How is Inverted Dropout implemented?

- We define a random binary mask $\boldsymbol{m}^{(l)}$ which is used to remove neurons and is generated by sampling a Bernoulli distribution with $P(x = 1) = p$, and note, $m^{(l)}$ changes for each iteration of the backpropagation algorithm.
- The forward pass of a Dropout layer (function) during **training** is given by $f(\boldsymbol{x}) = \boldsymbol{x} \odot \boldsymbol{m}/p$.
- The forward pass of a Dropout layer (function) during **inference** is given by $f(\boldsymbol{x}) = \boldsymbol{x}$.
- This can be applied to any layer(s) of the network *except the output layer!*
- It's not common to put it everywhere; just a couple of select places (empirically chosen).
- The gradient (during training) is simply the hadamard product of the incoming gradient with $\boldsymbol{m}/p$.

Why Does Dropout Work?

- Neurons cannot co-adapt to other units (they cannot assume that all of the other units will be present).
- By breaking co-adaptation, each unit will ultimately find more general features.

Why Does Dropout Work?

- Neurons cannot co-adapt to other units (they cannot assume that all of the other units will be present).
- By breaking co-adaptation, each unit will ultimately find more general features.
- By ensembling (averaging) multiple networks, each relying on different (but overlapping) features we get a more effective machine.