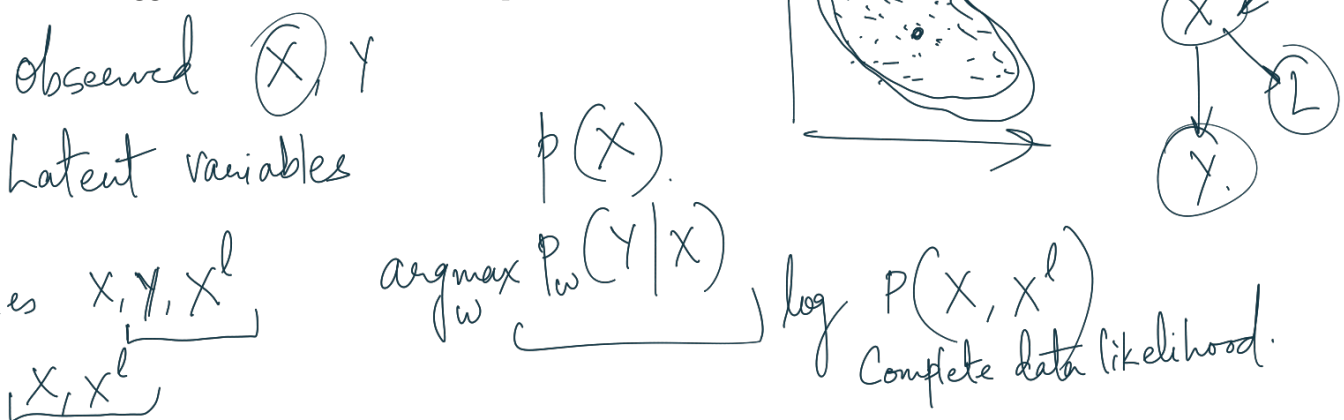
Chapter 3

Bayesian Deep Learning

In previous chapters we reviewed Bayesian neural networks (BNNs) and historical techniques for approximate inference in these, as well as more recent approaches. We discussed the advantages and disadvantages of different techniques, examining their practicality. This, perhaps, is the most important aspect of modern techniques for approximate inference in BNNs. The field of deep learning is pushed forward by practitioners, working on real-world problems. Techniques which cannot scale to complex models with potentially millions of parameters, scale well with large amounts of data, need well studied models to be radically changed, or are not accessible to engineers, will simply perish.

In this chapter we will develop on the strand of work of [Graves, 2011; Hinton and Van Camp, 1993], but will do so from the Bayesian perspective rather than the information theory one. Developing Bayesian approaches to deep learning, we will tie approximate BNN inference together with deep learning stochastic regularisation techniques (SRTs) such as dropout. These regularisation techniques are used in many modern deep learning tools, allowing us to offer a practical inference technique.

We will start by reviewing in detail the tools used by Graves [2011], and extend on these with recent research. In the process we will comment and analyse the variance of several stochastic estimators used in variational inference (VI). Following that we will tie these derivations to SRTs, and propose practical techniques to obtain model uncertainty, even from existing models. We finish the chapter by developing specific examples for image based models (CNNs) and sequence based models (RNNs). These will be demonstrated in chapter 5, where we will survey recent research making use of the suggested tools in real-world problems.



3.1 Advanced techniques in variational inference

We start by reviewing recent advances in VI. We are interested in the posterior over the weights given our observables $\mathbf{X}, \mathbf{Y}$: $p(\boldsymbol{\omega}|\mathbf{X}, \mathbf{Y})$. This posterior is not tractable for a Bayesian NN, and we use variational inference to approximate it. Recall our minimisation objective (eq. (2.6)),

$$\mathcal{L}_{\text{VI}}(\theta) := -\sum_{i=1}^{N} \int q_\theta(\boldsymbol{\omega}) \log p(\mathbf{y}_i|\mathbf{f}^{\boldsymbol{\omega}}(\mathbf{x}_i)) \mathrm{d}\boldsymbol{\omega} + \mathrm{KL}(q_\theta(\boldsymbol{\omega})||p(\boldsymbol{\omega})). \tag{3.1}$$

Evaluating this objective poses several difficulties. First, the summed-over terms $\int q_\theta(\boldsymbol{\omega}) \log p(\mathbf{y}_i|\mathbf{f}^{\boldsymbol{\omega}}(\mathbf{x}_i)) \mathrm{d}\boldsymbol{\omega}$ are not tractable for BNNs with more than a single hidden layer. Second, this objective requires us to perform computations over the entire dataset, which can be too costly for large N.

To solve the latter problem, we may use data sub-sampling (also referred to as *mini-batch optimisation*). We approximate eq. (3.1) with

$$\widehat{\mathcal{L}}_{\text{VI}}(\theta) := -\frac{N}{M} \sum_{i \in S} \int q_\theta(\boldsymbol{\omega}) \log p(\mathbf{y}_i|\mathbf{f}^{\boldsymbol{\omega}}(\mathbf{x}_i)) \mathrm{d}\boldsymbol{\omega} + \mathrm{KL}(q_\theta(\boldsymbol{\omega})||p(\boldsymbol{\omega})) \tag{3.2}$$

with a random index set S of size M.

The data sub-sampling approximation forms an unbiased stochastic estimator to eq. (3.1), meaning that $\mathbb{E}_S[\widehat{\mathcal{L}}_{\text{VI}}(\theta)] = \mathcal{L}_{\text{VI}}(\theta)$. We can thus use a stochastic optimiser to optimise this stochastic objective, and obtain a (local) optimum θ^* which would be an optimum to $\mathcal{L}_{\text{VI}}(\theta)$ as well [Robbins and Monro, 1951]. This approximation is often used in the deep learning literature, and was suggested by Hoffman et al. [2013] in the VI context (surprisingly) only recently[1].

The remaining difficulty with the objective in eq. (3.1) is the evaluation of the expected log likelihood $\int q_\theta(\boldsymbol{\omega}) \log p(\mathbf{y}_i|\mathbf{f}^{\boldsymbol{\omega}}(\mathbf{x}_i)) \mathrm{d}\boldsymbol{\omega}$. Monte Carlo integration of the integral has been attempted by some, to varying degrees of success. We will review three approaches used in the VI literature and analyse their variance.

3.1.1 Monte Carlo estimators in variational inference

We often use Monte Carlo (MC) estimation in variational inference to estimate the expected log likelihood (the integral in eq. (3.2)). But more importantly we wish to estimate the expected log likelihood's derivatives w.r.t. the approximating distribution

[1] Although online VI methods processing one point at a time have a long history [Ghahramani and Attias, 2000; Sato, 2001]

parameters θ. This allows us to optimise the objective and find the optimal parameters θ^*. There exist three main techniques for MC estimation in the VI literature (a brief survey of the literature was collected by [Schulman et al., 2015]). These have very different characteristics and variances for the estimation of the expected log likelihood and its derivative. Here we will contrast all three techniques in the context of VI and analyse them both empirically and theoretically.

To present the various techniques we will consider the general case of estimating the *integral derivative*:

$$I(\theta) = \frac{\partial}{\partial\theta}\int f(x)p_\theta(x)\mathrm{d}x \tag{3.3}$$

which arises when we optimise eq. (3.2) (we will also refer to a stochastic estimator of this quantity as a *stochastic derivative estimator*). Here $f(x)$ is a function defined on the reals, differentiable almost everywhere (a.e., differentiable on $\mathbb{R}$ apart from a zero measure set), and $p_\theta(x)$ is a probability density function (pdf) parametrised by θ from which we can easily generate samples. We assume that the integral exists and is finite, and that $f(x)$ does not depend on θ. Note that we shall write $f'(x)$ when we differentiate $f(x)$ w.r.t. its input (i.e. $\frac{\partial}{\partial x}f(x)$, in contrast to the differentiation of $f(x)$ w.r.t. other variables).

We further use $p_\theta(x) = \mathcal{N}(x; \mu, \sigma^2)$ as a concrete example, with $\theta = \{\mu, \sigma\}$. In this case, we will refer to an estimator of (3.3) as the *mean derivative estimator* (for some function $f(x)$) when we differentiate w.r.t. $\theta = \mu$, and the *standard deviation derivative estimator* when we differentiate w.r.t. $\theta = \sigma$.

Three MC estimators for eq. (3.3) are used in the VI literature:

1. The *score function* estimator (also known as a *likelihood ratio estimator* and *Reinforce*, [Fu, 2006; Glynn, 1990; Paisley et al., 2012; Williams, 1992]) relies on the identity $\frac{\partial}{\partial\theta}p_\theta(x) = p_\theta(x)\frac{\partial}{\partial\theta}\log p_\theta(x)$ and follows the parametrisation:

$$\begin{aligned}\frac{\partial}{\partial\theta}\int f(x)p_\theta(x)\mathrm{d}x &= \int f(x)\frac{\partial}{\partial\theta}p_\theta(x)\mathrm{d}x \\ &= \int f(x)\frac{\partial \log p_\theta(x)}{\partial\theta}p_\theta(x)\mathrm{d}x\end{aligned} \tag{3.4}$$

leading to the unbiased stochastic estimator $\widehat{I}_1(\theta) = f(x)\frac{\partial \log p_\theta(x)}{\partial\theta}$ with $x \sim p_\theta(x)$, hence $\mathbb{E}_{p_\theta(x)}[\widehat{I}_1(\theta)] = I(\theta)$. Note that the first transition is possible since x and $f(x)$ do not depend on θ, and only $p_\theta(x)$ depends on it. This estimator is simple and applicable with discrete distributions, but as Paisley et al. [2012] identify, it

has rather high variance. When used in practice it is often coupled with a variance reduction technique.

2. Eq. (3.3) can be re-parametrised to obtain an alternative MC estimator, which we refer to as a *pathwise derivative* estimator (this estimator is also referred to in the literature as the *re-parametrisation trick*, *infinitesimal perturbation analysis*, and *stochastic backpropagation* [Glasserman, 2013; Kingma and Welling, 2013, 2014; Rezende et al., 2014; Titsias and Lázaro-Gredilla, 2014]). Assume that $p_\theta(x)$ can be re-parametrised as $p(\epsilon)$, a parameter-free distribution, s.t. $x = g(\theta, \epsilon)$ with a deterministic *differentiable* bivariate transformation $g(\cdot, \cdot)$. For example, with our $p_\theta(x) = \mathcal{N}(x; \mu, \sigma^2)$ we have $g(\theta, \epsilon) = \mu + \sigma\epsilon$ together with $p(\epsilon) = \mathcal{N}(\epsilon; 0, I)$. Then the following estimator arises:

$$\widehat{I}_2(\theta) = f'(g(\theta, \epsilon)) \frac{\partial}{\partial \theta} g(\theta, \epsilon). \tag{3.5}$$

Here $\mathbb{E}_{p(\epsilon)}[\widehat{I}_2(\theta)] = I(\theta)$.

Note that compared to the estimator above where $f(x)$ is used, in this case we use its derivative $f'(x)$. In the Gaussian case, both the derivative w.r.t. μ as well as the derivative w.r.t. σ use $f(x)$'s first derivative (note the substitution of ϵ with $x = \mu + \sigma\epsilon$):

$$\begin{aligned} \frac{\partial}{\partial \mu} \int f(x) p_\theta(x) \mathrm{d}x &= \int f'(x) p_\theta(x) \mathrm{d}x, \\ \frac{\partial}{\partial \sigma} \int f(x) p_\theta(x) \mathrm{d}x &= \int f'(x) \frac{(x-\mu)}{\sigma} p_\theta(x) \mathrm{d}x, \end{aligned}$$

i.e. $\widehat{I}_2(\mu) = f'(x)$ and $\widehat{I}_2(\sigma) = f'(x)\frac{(x-\mu)}{\sigma}$ with $\mathbb{E}_{p_\theta(x)}[\widehat{I}_2(\mu)] = I(\mu)$ and $\mathbb{E}_{p_\theta(x)}[\widehat{I}_2(\sigma)] = I(\sigma)$. An interesting question arises when we compare this estimator ($\widehat{I}_2(\theta)$) to the one above ($\widehat{I}_1(\theta)$): why is it that in one case we use the function $f(x)$, and in the other use its derivative? This is answered below, together with an alternative derivation to that of Kingma and Welling [2013, section 2.4].

The pathwise derivative estimator seems to (empirically) result in a lower variance estimator than the one above, although to the best of my knowledge no proof to this has been given in the literature. An analysis of this parametrisation, together with examples where the estimator has *higher* variance than that of the score function estimator is given below.

3. Lastly, it is important to mention the estimator given in [Opper and Archambeau, 2009, eq. (6-7)] (studied in [Rezende et al., 2014] as well). Compared to both estimators above, Opper and Archambeau [2009] relied on the characteristic function of the Gaussian distribution, restricting the estimator to Gaussian $p_\theta(x)$ alone. The resulting mean derivative estimator, $\frac{\partial}{\partial \mu} \int f(x) p_\theta(x) \mathrm{d}x$, is identical to the one resulting from the pathwise derivative estimator.

   But when differentiating w.r.t. σ, the estimator depends on $f(x)$'s second derivative [Opper and Archambeau, 2009, eq. (19)]:

$$\frac{\partial}{\partial \sigma} \int f(x) p_\theta(x) \mathrm{d}x = 2\sigma \cdot \frac{1}{2} \int f''(x) p_\theta(x) \mathrm{d}x \tag{3.6}$$

   i.e. $\widehat{I}_3(\sigma) = \sigma f''(x)$ with $\mathbb{E}_{p_\theta(x)}[\widehat{I}_3(\sigma)] = I(\sigma)$. This is in comparison to the estimators above that use $f(x)$ or its first derivative. We refer to this estimator as a *characteristic function* estimator.

These three MC estimators are believed to have decreasing estimator variances (1) > (2) > (3). Before we analyse this variance, we offer an alternative derivation to Kingma and Welling [2013]'s derivation of the pathwise derivative estimator.

Remark (Auxiliary variable view of the pathwise derivative estimator)**.** Kingma and Welling [2013]'s derivation of the pathwise derivative estimator was obtained through a change of variables. Instead we justify the estimator through a construction relying on an auxiliary variable augmenting the distribution $p_\theta(x)$.
Assume that the distribution $p_\theta(x)$ can be written as $\int p_\theta(x, \epsilon) \mathrm{d}\epsilon = \int p_\theta(x|\epsilon) p(\epsilon) \mathrm{d}\epsilon$, with $p_\theta(x|\epsilon) = \delta(x - g(\theta, \epsilon))$ a dirac delta function. Then

$$\begin{aligned}
\frac{\partial}{\partial \theta} \int f(x) p_\theta(x) \mathrm{d}x &= \frac{\partial}{\partial \theta} \int f(x) \Big(\int p_\theta(x, \epsilon) \mathrm{d}\epsilon \Big) \mathrm{d}x \\
&= \frac{\partial}{\partial \theta} \int f(x) p_\theta(x|\epsilon) p(\epsilon) \mathrm{d}\epsilon \mathrm{d}x \\
&= \frac{\partial}{\partial \theta} \int \Big(\int f(x) \delta\big(x - g(\theta, \epsilon)\big) \mathrm{d}x \Big) p(\epsilon) \mathrm{d}\epsilon.
\end{aligned}$$

Now, since $\delta(x - g(\theta, \epsilon))$ is zero for all x apart from $x = g(\theta, \epsilon)$,

$$\frac{\partial}{\partial \theta} \int \Big(\int f(x) \delta\big(x - g(\theta, \epsilon)\big) \mathrm{d}x \Big) p(\epsilon) \mathrm{d}\epsilon = \frac{\partial}{\partial \theta} \int f(g(\theta, \epsilon)) p(\epsilon) \mathrm{d}\epsilon$$

$$= \int \frac{\partial}{\partial \theta} f(g(\theta, \epsilon)) p(\epsilon) \mathrm{d}\epsilon$$
$$= \int f'(g(\theta, \epsilon)) \frac{\partial}{\partial \theta} g(\theta, \epsilon) p(\epsilon) \mathrm{d}\epsilon.$$

This derivation raises an interesting question: why is it that here the function $f(x)$ depends on θ, whereas in the above (the score function estimator) it does not? A clue into explaining this can be found through a measure theoretic view of the score function estimator:

$$\frac{\partial}{\partial \theta} \int f(x) p_\theta(x) \mathrm{d}x = \frac{\partial}{\partial \theta} \int \Big(f(x) p_\theta(x) \Big) \mathrm{d}\lambda(x)$$

where $\lambda(x)$ is the Lebesgue measure. Here the measure does not depend on θ, hence x does not depend on θ. Only the integrand $f(x)p_\theta(x)$ depends on θ, therefore the estimator depends on $\frac{\partial}{\partial \theta} p_\theta(x)$ and $f(x)$. This is in comparison to the pathwise derivative estimator:

$$\frac{\partial}{\partial \theta} \int f(x) p_\theta(x) \mathrm{d}x = \frac{\partial}{\partial \theta} \int f(x) \mathrm{d}p_\theta(x).$$

Here the integration is w.r.t. the measure $p_\theta(x)$, which depends on θ. As a result the random variable x as a measurable function depends on θ, leading to $f(x)$ being a measurable function depending on θ. Intuitively, the above can be seen as "stretching" and "contracting" the space following the density $p_\theta(x)$, leading to a function defined on this space to depend on θ.

3.1.2 Variance analysis of Monte Carlo estimators in variational inference

Next we analyse the estimator variance for the three estimators above using a Gaussian distribution $p_\theta(x) = \mathcal{N}(x; \mu, \sigma^2)$. We will refer to the estimators as the score function estimator (1), the pathwise derivative estimator (2), and the characteristic function estimator (3). We will alternate between the estimator names and numbers indistinguishably.

We begin with the observation that none of the estimators has the lowest variance for *all* functions $f(x)$. For each estimator there exists a function $f(x)$ such that the estimator would achieve lowest variance when differentiating w.r.t. μ, $\frac{\partial}{\partial \mu} \int f(x) p_\theta(x) \mathrm{d}x$. Further, for each estimator there exists a function $f(x)$ (not necessarily the same) such that the estimator would achieve lowest variance when differentiating w.r.t. σ, $\frac{\partial}{\partial \sigma} \int f(x) p_\theta(x) \mathrm{d}x$.

$f(x)$	Score function	Pathwise derivative	Character. function
$x + x^2$	$1.7 \cdot 10^1$	$\mathbf{4.0 \cdot 10^0}$	$\mathbf{4.0 \cdot 10^0}$
$\sin(x)$	$3.3 \cdot 10^{-1}$	$\mathbf{2.0 \cdot 10^{-1}}$	$\mathbf{2.0 \cdot 10^{-1}}$
$\sin(10x)$	$\mathbf{5.0 \cdot 10^{-1}}$	$5.0 \cdot 10^1$	$5.0 \cdot 10^1$

Table 3.1 $\frac{\partial}{\partial \boldsymbol{\mu}} \int f(x) p_\theta(x) \mathrm{d}x$ variance

$f(x)$	Score function	Pathwise derivative	Character. function
$x + x^2$	$8.6 \cdot 10^1$	$9.1 \cdot 10^0$	$\mathbf{0.0 \cdot 10^0}$
$\sin(x)$	$8.7 \cdot 10^{-1}$	$\mathbf{3.0 \cdot 10^{-1}}$	$4.3 \cdot 10^{-1}$
$\sin(10x)$	$\mathbf{1.0 \cdot 10^0}$	$5.0 \cdot 10^1$	$5.0 \cdot 10^3$

Table 3.2 $\frac{\partial}{\partial \boldsymbol{\sigma}} \int f(x) p_\theta(x) \mathrm{d}x$ variance

Table 3.3 Estimator variance for various functions $f(x)$ for the score function estimator, the pathwise derivative estimator, and the characteristic function estimator ((1), (2), and (3) above). On the left is mean derivative estimator variance, and on the right is standard deviation derivative estimator variance, both w.r.t. $p_\theta = \mathcal{N}(\mu, \sigma^2)$ and evaluated at $\mu = 0, \sigma = 1$. In bold is lowest estimator variance.

We assess empirical sample variance of $T = 10^6$ samples for the mean and standard deviation derivative estimators. Tables 3.1 and 3.2 show estimator sample variance for the integral derivative w.r.t. μ and σ respectively, for three different functions $f(x)$. Even though all estimators result in roughly the same means, their variances differ considerably.

For functions with slowly varying derivatives in a neighbourhood of zero (such as the smooth function $f(x) = x + x^2$) we have that the estimator variance obeys (1) > (2) > (3). Also note that mean variance for (2) and (3) are identical, and that σ derivative variance under the characteristic function estimator in this case is zero (since the second derivative for this function is zero). Lastly, note that even though $f(x) = \sin(x)$ is smooth with bounded derivatives, the similar function $f(x) = \sin(10x)$ has variance (3) > (2) > (1). This is because the derivative of the function has high magnitude and varies much near zero.

I will provide a simple property a function has to satisfy for it to have lower variance under the pathwise derivative estimator and the characteristic function estimator than the score function estimator. This will be for the mean derivative estimator.

Proposition 1. *Let $f(x)$, $f'(x)$, $f''(x)$ be real-valued functions s.t. $f(x)$ is an indefinite integral of $f'(x)$, and $f'(x)$ is an indefinite integral of $f''(x)$. Assume that $Var_{p_\theta(x)}((x-\mu)f(x)) < \infty$, and $Var_{p_\theta(x)}(f'(x)) < \infty$, as well as $\mathbb{E}_{p_\theta(x)}(|(x-\mu)f'(x) + f(x)|) < \infty$ and $\mathbb{E}_{p_\theta(x)}(|f''(x)|) < \infty$, with $p_\theta(x) = \mathcal{N}(\mu, \sigma^2)$.*

If it holds that

$$\mathbb{E}_{p_\theta(x)}\Big((x-\mu)f'(x) + f(x)\Big)^2 - \sigma^4 \mathbb{E}_{p_\theta(x)}\Big(f''(x)^2\Big) \geq 0,$$

then the pathwise derivative and the characteristic function mean derivative estimators w.r.t. the function $f(x)$ will have lower variance than the score function estimator.

Before proving the proposition, I will give some intuitive insights into what the condition means. For this, assume for simplicity that $\mu = 0$ and $\sigma = 1$. This gives the simplified condition

$$\mathbb{E}_{p_\theta(x)}\Big(xf'(x) + f(x)\Big)^2 \geq \mathbb{E}_{p_\theta(x)}\Big(f''(x)^2\Big).$$

First, observe that all expectations are taken over $p_\theta(x)$, meaning that we only care about the functions' average behaviour near zero. Second, a function $f(x)$ with a large derivative absolute value will change considerably, hence will have high variance. Since the pathwise derivative estimator boils down to $f'(x)$ in the Gaussian case, and the score function estimator boils down to $xf(x)$ (shown in the proof), we wish the expected change in $\frac{\partial}{\partial x}xf(x) = xf'(x) + f(x)$ to be higher than the expected change in $\frac{\partial}{\partial x}f'(x) = f''(x)$, hence the condition.

Proof. We start with the observation that the score function mean estimator w.r.t. a Gaussian $p_\theta(x)$ is given by

$$\frac{\partial}{\partial \mu}\int f(x)p_\theta(x)\mathrm{d}x = \int f(x)\frac{x-\mu}{\sigma^2}p_\theta(x)\mathrm{d}x$$

resulting in the estimator $\widehat{I}_1 = f(x)\frac{x-\mu}{\sigma^2}$. The pathwise derivative and the characteristic function estimators are identical for the mean derivative, and given by $\widehat{I}_2 = f'(x)$. We thus wish to show that $\mathrm{Var}_{p_\theta(x)}(\widehat{I}_2) \leq \mathrm{Var}_{p_\theta(x)}(\widehat{I}_1)$ under the proposition's assumptions. Since $\mathrm{Var}_{p_\theta(x)}(\widehat{I}_1) = \frac{\mathrm{Var}_{p_\theta(x)}(xf(x)-\mu f(x))}{\sigma^4}$, we wish to show that

$$\sigma^4\mathrm{Var}_{p_\theta(x)}\Big(f'(x)\Big) \leq \mathrm{Var}_{p_\theta(x)}\Big((x-\mu)f(x)\Big).$$

Proposition 3.2 in [Cacoullos, 1982] states that for $g(x)$, $g'(x)$ real-valued functions s.t. $g(x)$ is an indefinite integral of $g'(x)$, and $\mathrm{Var}_{p_\theta(x)}(g(x)) < \infty$ and $\mathbb{E}_{p_\theta(x)}(|g'(x)|) < \infty$, there exists that

$$\sigma^2\mathbb{E}_{p_\theta(x)}\Big(g'(x)\Big)^2 \leq \mathrm{Var}_{p_\theta(x)}\Big(g(x)\Big) \leq \sigma^2\mathbb{E}_{p_\theta(x)}\Big(g'(x)^2\Big).$$

Substituting $g(x)$ with $(x-\mu)f(x)$ we have

$$\sigma^2\mathbb{E}_{p_\theta(x)}\Big((x-\mu)f'(x) + f(x)\Big)^2 \leq \mathrm{Var}_{p_\theta(x)}\Big((x-\mu)f(x)\Big)$$

and substituting $g(x)$ with $f'(x)$ we have

$$\mathrm{Var}_{p_\theta(x)}\Big(f'(x)\Big) \leq \sigma^2 \mathbb{E}_{p_\theta(x)}\Big(f''(x)^2\Big).$$

Under the proposition's assumption that

$$\mathbb{E}_{p_\theta(x)}\Big((x-\mu)f'(x)+f(x)\Big)^2 - \sigma^4 \mathbb{E}_{p_\theta(x)}\Big(f''(x)^2\Big) \geq 0$$

we conclude

$$\begin{aligned}\sigma^4 \mathrm{Var}_{p_\theta(x)}\Big(f'(x)\Big) \leq \sigma^6 \mathbb{E}_{p_\theta(x)}\Big(f''(x)^2\Big) &\leq \sigma^2 \mathbb{E}_{p_\theta(x)}\Big((x-\mu)f'(x)+f(x)\Big)^2 \\ &\leq \mathrm{Var}_{p_\theta(x)}\Big((x-\mu)f(x)\Big)\end{aligned}$$

as we wanted to show. □

For example, with the simple polynomial function $f(x) = x + x^2$ and $\mu = 0, \sigma = 1$ from table 3.3 we have

$$\begin{aligned}\mathbb{E}_{\mathcal{N}(0,1)}\Big(xf'(x)+f(x)\Big)^2 - \mathbb{E}_{\mathcal{N}(0,1)}\Big(f''(x)^2\Big) &= \mathbb{E}_{\mathcal{N}(0,1)}\Big(2x+3x^2\Big)^2 - 2^2 \\ &= 3^2 - 2^2 > 0,\end{aligned}$$

and indeed the score function estimator variance is higher than that of the pathwise derivative estimator and that of the characteristic function estimator. A similar result can be derived for the standard deviation derivative estimator.

From empirical observation, the functions $f(x)$ often encountered in VI seem to satisfy the variance relation (1) > (2). For this reason, and since we will make use of distributions other than Gaussian, we continue our work using the pathwise derivative estimator.

3.2 Practical inference in Bayesian neural networks

We now derive what would hopefully be a practical inference method for Bayesian neural networks. Inspired by the work of Graves [2011], we propose an inference technique that satisfies our definition of *practicality*, making use of the tools above. The work in this section and the coming sections was previously presented in [Gal, 2015; Gal and Ghahramani, 2015a,b,c,d, 2016a,b,c].

In his work, Graves [2011] used both delta approximating distributions, as well as fully factorised Gaussian approximating distributions. As such, Graves [2011] relied on Opper and Archambeau [2009]'s characteristic function estimator in his approximation of eq. (3.2). Further, Graves [2011] factorised the approximating distribution for each weight scalar, losing weight correlations. This approach has led to the limitations discussed in §2.2.2, hurting the method's performance and practicality.

Using the tools above, and relying on the pathwise derivative estimator instead of the characteristic function estimator in particular, we can make use of more interesting non-Gaussian approximating distributions. Further, to avoid losing weight correlations, we factorise the distribution for each weight row $\mathbf{w}_{l,i}$ in each weight matrix $\mathbf{W}_l$, instead of factorising over each *weight scalar*. The reason for this will be given below. Using these two key changes, we will see below how our approximate inference can be closely tied to SRTs, suggesting a practical, well performing, implementation.

To use the pathwise derivative estimator we need to re-parametrise each $q_{\theta_{l,i}}(\mathbf{w}_{l,i})$ as $\mathbf{w}_{l,i} = g(\theta_{l,i}, \epsilon_{l,i})$ and specify some $p(\epsilon_{l,i})$ (this will be done at a later time). For simplicity of notation we will write $p(\boldsymbol{\epsilon}) = \prod_{l,i} p(\epsilon_{l,i})$, and $\boldsymbol{\omega} = g(\theta, \boldsymbol{\epsilon})$ collecting all model random variables. Starting from the data sub-sampling objective (eq. (3.2)), we re-parametrise each integral to integrate w.r.t. $p(\boldsymbol{\epsilon})$:

$$\begin{aligned}\widehat{\mathcal{L}}_{\text{VI}}(\theta) &= -\frac{N}{M}\sum_{i\in S}\int q_\theta(\boldsymbol{\omega}) \log p(\mathbf{y}_i|\mathbf{f}^{\boldsymbol{\omega}}(\mathbf{x}_i))\mathrm{d}\boldsymbol{\omega} + \mathrm{KL}(q_\theta(\boldsymbol{\omega})||p(\boldsymbol{\omega})) \\ &= -\frac{N}{M}\sum_{i\in S}\int p(\boldsymbol{\epsilon}) \log p(\mathbf{y}_i|\mathbf{f}^{g(\theta,\boldsymbol{\epsilon})}(\mathbf{x}_i))\mathrm{d}\boldsymbol{\epsilon} + \mathrm{KL}(q_\theta(\boldsymbol{\omega})||p(\boldsymbol{\omega}))\end{aligned}$$

and then replace each expected log likelihood term with its stochastic estimator (eq. (3.5)), resulting in a new MC estimator:

$$\widehat{\mathcal{L}}_{\text{MC}}(\theta) = -\frac{N}{M}\sum_{i\in S} \log p(\mathbf{y}_i|\mathbf{f}^{g(\theta,\boldsymbol{\epsilon})}(\mathbf{x}_i)) + \mathrm{KL}(q_\theta(\boldsymbol{\omega})||p(\boldsymbol{\omega})) \tag{3.7}$$

s.t. $\mathbb{E}_{S,\boldsymbol{\epsilon}}(\widehat{\mathcal{L}}_{\text{MC}}(\theta)) = \mathcal{L}_{\text{VI}}(\theta)$.

Following results in stochastic non-convex optimisation [Rubin, 1981], optimising $\widehat{\mathcal{L}}_{\text{MC}}(\theta)$ w.r.t. θ would converge to the same optima as optimising our original objective $\mathcal{L}_{\text{VI}}(\theta)$. One thus follows algorithm 1 for inference.

Predictions with this approximation follow equation (2.4) which replaces the posterior $p(\boldsymbol{\omega}|\mathbf{X}, \mathbf{Y})$ with the approximate posterior $q_\theta(\boldsymbol{\omega})$. We can then approximate the predictive

Algorithm 1 Minimise divergence between $q_\theta(\boldsymbol{\omega})$ and $p(\boldsymbol{\omega}|X, Y)$

1: Given dataset $\mathbf{X}, \mathbf{Y}$,
2: Define learning rate schedule η,
3: Initialise parameters θ randomly.
4: **repeat**
5: Sample M random variables $\widehat{\boldsymbol{\epsilon}}_i \sim p(\boldsymbol{\epsilon})$, S a random subset of $\{1, .., N\}$ of size M.
6: Calculate stochastic derivative estimator w.r.t. θ:

$$\widehat{\Delta\theta} \leftarrow -\frac{N}{M}\sum_{i\in S}\frac{\partial}{\partial\theta}\log p(\mathbf{y}_i|\mathbf{f}^{g(\theta,\widehat{\boldsymbol{\epsilon}}_i)}(\mathbf{x}_i)) + \frac{\partial}{\partial\theta}\mathrm{KL}(q_\theta(\boldsymbol{\omega})||p(\boldsymbol{\omega})).$$

7: Update θ:

$$\theta \leftarrow \theta + \eta\widehat{\Delta\theta}.$$

8: **until** θ has converged.

distribution with MC integration as well:

$$\begin{aligned}\tilde{q}_\theta(\mathbf{y}^*|\mathbf{x}^*) := \frac{1}{T}\sum_{t=1}^{T} p(\mathbf{y}^*|\mathbf{x}^*, \widehat{\boldsymbol{\omega}}_t) &\xrightarrow[T\to\infty]{} \int p\big(\mathbf{y}^*|\mathbf{x}^*, \boldsymbol{\omega}\big)q_\theta(\boldsymbol{\omega})\mathrm{d}\boldsymbol{\omega} \\ &\approx \int p\big(\mathbf{y}^*|\mathbf{x}^*, \boldsymbol{\omega}\big)p(\boldsymbol{\omega}|\mathbf{X}, \mathbf{Y})\mathrm{d}\boldsymbol{\omega} \\ &= p(\mathbf{y}^*|\mathbf{x}^*, \mathbf{X}, \mathbf{Y})\end{aligned} \tag{3.8}$$

with $\widehat{\boldsymbol{\omega}}_t \sim q_\theta(\boldsymbol{\omega})$.

We next present distributions $q_\theta(\boldsymbol{\omega})$ corresponding to several SRTs, s.t. standard techniques in the deep learning literature could be seen as identical to executing algorithm 1 for approximate inference with $q_\theta(\boldsymbol{\omega})$. This means that existing models that use such SRTs can be interpreted as performing approximate inference. As a result, uncertainty information can be extracted from these, as we will see in the following sections.

3.2.1 Stochastic regularisation techniques

First, what are SRTs? Stochastic regularisation techniques are techniques used to regularise deep learning models through the injection of stochastic noise into the model. By far the most popular technique is *dropout* [Hinton et al., 2012; Srivastava et al., 2014], but other techniques exist such as multiplicative Gaussian noise (MGN, also referred to as *Gaussian dropout*) [Srivastava et al., 2014], or dropConnect [Wan et al., 2013], among many others [Huang et al., 2016; Krueger et al., 2016; Moon et al., 2015; Singh et al.,

2016]. We will concentrate on dropout for the moment, and discuss alternative SRTs below.

Notation remark. In this section and the next, in order to avoid confusion between matrices (used as weights in a NN) and stochastic random matrices (which are random variables inducing a distribution over BNN weights), we change our notation slightly from §1.1. Here we use $\mathbf{M}$ to denote a *deterministic* matrix over the reals, $\mathbf{W}$ to denote a *random variable* defined over the set of real matrices[2], and use $\widehat{\mathbf{W}}$ to denote a realisation of $\mathbf{W}$.

Dropout is a technique used to avoid over-fitting in neural networks. It was suggested as an ad-hoc technique, and was motivated with sexual breeding metaphors rather than through theoretical foundations [Srivastava et al., 2014, page 1932]. It was introduced several years ago by Hinton et al. [2012] and studied more extensively in [Srivastava et al., 2014]. We will describe the use of dropout in simple single hidden layer neural networks (following the notation of §1.1 with the adjustments above). To use dropout we sample two binary vectors $\widehat{\boldsymbol{\epsilon}}_1, \widehat{\boldsymbol{\epsilon}}_2$ of dimensions Q (input dimensionality) and K (intermediate layer dimensionality) respectively. The elements of the vector $\widehat{\boldsymbol{\epsilon}}_i$ take value 0 with probability $0 \leq p_i \leq 1$ for $i = 1, 2$. Given an input $\mathbf{x}$, we set p_1 proportion of the elements of the input to zero in expectation: $\widehat{\mathbf{x}} = \mathbf{x} \odot \widehat{\boldsymbol{\epsilon}}_1$[3]. The output of the first layer is given by $\mathbf{h} = \sigma(\widehat{\mathbf{x}}\mathbf{M}_1 + \mathbf{b})$, in which we randomly set p_2 proportion of the elements to zero: $\widehat{\mathbf{h}} = \mathbf{h} \odot \widehat{\boldsymbol{\epsilon}}_2$, and linearly transform the vector to give the dropout model's output $\widehat{\mathbf{y}} = \widehat{\mathbf{h}}\mathbf{M}_2$. We repeat this for multiple layers.

We sample new realisations for the binary vectors $\widehat{\boldsymbol{\epsilon}}_i$ for every input point and every forward pass through the model (evaluating the model's output), and use the same values in the backward pass (propagating the derivatives to the parameters to be optimised $\theta = \{\mathbf{M}_1, \mathbf{M}_2, \mathbf{b}\}$). At test time we do not sample any variables and simply use the original units $\mathbf{x}, \mathbf{h}$ scaled by $\frac{1}{1-p_i}$.

Multiplicative Gaussian noise is similar to dropout, where the only difference is that $\widehat{\boldsymbol{\epsilon}}_i$ are vectors of draws from a Gaussian distribution $\mathcal{N}(1, \alpha)$ with a positive parameter α, rather than draws from a Bernoulli distribution.

[2]The reason for this notation is that $\mathbf{M}$ will often coincide with the mean of the random matrix $\mathbf{W}$.
[3]Here $\odot$ is the element-wise product.

3.2.2 Stochastic regularisation techniques as approximate inference

Dropout and most other SRTs view the injected noise as applied in the feature space (the input features to each layer: $\mathbf{x}, \mathbf{h}$). In Bayesian NNs, on the other hand, the stochasticity comes from our uncertainty over the model parameters. We can transform dropout's noise from the feature space to the parameter space as follows[4]:

$$\begin{aligned}
\widehat{\mathbf{y}} &= \widehat{\mathbf{h}}\mathbf{M}_2 \\
&= (\mathbf{h} \odot \widehat{\boldsymbol{\epsilon}}_2)\mathbf{M}_2 \\
&= (\mathbf{h} \cdot \text{diag}(\widehat{\boldsymbol{\epsilon}}_2))\mathbf{M}_2 \\
&= \mathbf{h}(\text{diag}(\widehat{\boldsymbol{\epsilon}}_2)\mathbf{M}_2) \\
&= \sigma\big(\widehat{\mathbf{x}}\mathbf{M}_1 + \mathbf{b}\big)(\text{diag}(\widehat{\boldsymbol{\epsilon}}_2)\mathbf{M}_2) \\
&= \sigma\big((\mathbf{x} \odot \widehat{\boldsymbol{\epsilon}}_1)\mathbf{M}_1 + \mathbf{b}\big)(\text{diag}(\widehat{\boldsymbol{\epsilon}}_2)\mathbf{M}_2) \\
&= \sigma\big(\mathbf{x}(\text{diag}(\widehat{\boldsymbol{\epsilon}}_1)\mathbf{M}_1) + \mathbf{b}\big)(\text{diag}(\widehat{\boldsymbol{\epsilon}}_2)\mathbf{M}_2)
\end{aligned}$$

writing $\widehat{\mathbf{W}}_1 := \text{diag}(\widehat{\boldsymbol{\epsilon}}_1)\mathbf{M}_1$ and $\widehat{\mathbf{W}}_2 := \text{diag}(\widehat{\boldsymbol{\epsilon}}_2)\mathbf{M}_2$ we end up with

$$\widehat{\mathbf{y}} = \sigma\big(\mathbf{x}\widehat{\mathbf{W}}_1 + \mathbf{b}\big)\widehat{\mathbf{W}}_2 =: \mathbf{f}^{\widehat{\mathbf{W}}_1, \widehat{\mathbf{W}}_2, \mathbf{b}}(\mathbf{x})$$

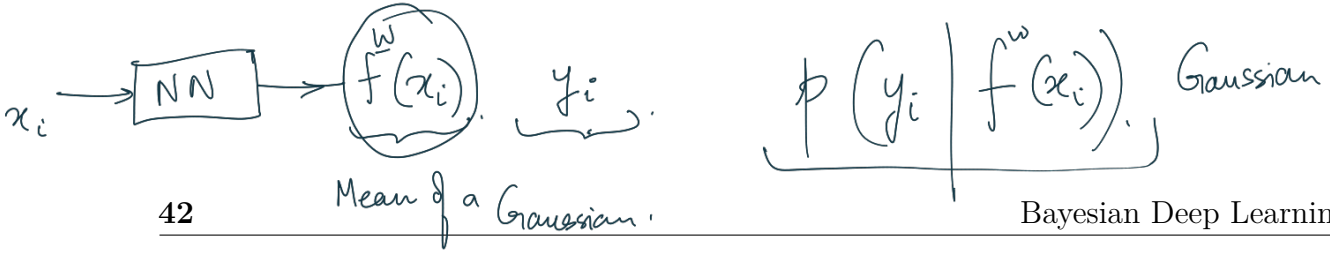
with random variable realisations as weights, and write $\widehat{\boldsymbol{\omega}} = \{\widehat{\mathbf{W}}_1, \widehat{\mathbf{W}}_2, \mathbf{b}\}$.

This allows us to write dropout's objective in a more convenient form. Recall that a neural network's optimisation objective is given by eq. (1.3). For dropout it will simply be:

$$\widehat{\mathcal{L}}_{\text{dropout}}(\mathbf{M}_1, \mathbf{M}_2, \mathbf{b}) := \frac{1}{M}\sum_{i \in S} E^{\widehat{\mathbf{W}}_1^i, \widehat{\mathbf{W}}_2^i, \mathbf{b}}(\mathbf{x}_i, \mathbf{y}_i) + \lambda_1 ||\mathbf{M}_1||^2 + \lambda_2 ||\mathbf{M}_2||^2 + \lambda_3 ||\mathbf{b}||^2, \tag{3.9}$$

with $\widehat{\mathbf{W}}_1^i, \widehat{\mathbf{W}}_2^i$ corresponding to new masks $\widehat{\boldsymbol{\epsilon}}_1^i, \widehat{\boldsymbol{\epsilon}}_2^i$ sampled for each data point i. Here we used data sub-sampling with a random index set S of size M, as is common in deep learning.

[4] Here the diag(·) operator maps a vector to a diagonal matrix whose diagonal is the elements of the vector.



As shown by Tishby et al. [1989], in regression $E^{\mathbf{M}_1,\mathbf{M}_2,\mathbf{b}}(\mathbf{x},\mathbf{y})$ can be rewritten as the negative log-likelihood scaled by a constant:

$$E^{\mathbf{M}_1,\mathbf{M}_2,\mathbf{b}}(\mathbf{x},\mathbf{y}) = \frac{1}{2}||\mathbf{y} - \mathbf{f}^{\mathbf{M}_1,\mathbf{M}_2,\mathbf{b}}(\mathbf{x})||^2 = -\frac{1}{\tau}\log p(\mathbf{y}|\mathbf{f}^{\mathbf{M}_1,\mathbf{M}_2,\mathbf{b}}(\mathbf{x})) + \text{const} \tag{3.10}$$

where $p(\mathbf{y}|\mathbf{f}^{\mathbf{M}_1,\mathbf{M}_2,\mathbf{b}}(\mathbf{x})) = \mathcal{N}(\mathbf{y}; \mathbf{f}^{\mathbf{M}_1,\mathbf{M}_2,\mathbf{b}}(\mathbf{x}), \tau^{-1}I)$ with τ^{-1} observation noise. It is simple to see that this holds for classification as well (in which case we should set $\tau = 1$).

Recall that $\widehat{\boldsymbol{\omega}} = \{\widehat{\mathbf{W}}_1, \widehat{\mathbf{W}}_2, \mathbf{b}\}$ and write

$$\widehat{\boldsymbol{\omega}}_i = \{\widehat{\mathbf{W}}_1^i, \widehat{\mathbf{W}}_2^i, \mathbf{b}\} = \{\text{diag}(\widehat{\boldsymbol{\epsilon}}_1^i)\mathbf{M}_1, \text{diag}(\widehat{\boldsymbol{\epsilon}}_2^i)\mathbf{M}_2, \mathbf{b}\} =: g(\theta, \widehat{\boldsymbol{\epsilon}}_i)$$

with $\theta = \{\mathbf{M}_1, \mathbf{M}_2, \mathbf{b}\}$, $\widehat{\boldsymbol{\epsilon}}_1^i \sim p(\boldsymbol{\epsilon}_1)$, and $\widehat{\boldsymbol{\epsilon}}_2^i \sim p(\boldsymbol{\epsilon}_2)$ for $1 \leq i \leq N$. Here $p(\boldsymbol{\epsilon}_l)$ $(l = 1, 2)$ is a product of Bernoulli distributions with probabilities $1 - p_l$, from which a realisation would be a vector of zeros and ones.

We can plug identity (3.10) into objective (3.9) and get

$$\widehat{\mathcal{L}}_{\text{dropout}}(\mathbf{M}_1, \mathbf{M}_2, \mathbf{b}) = -\frac{1}{M\tau}\sum_{i\in S}\log p(\mathbf{y}_i|\mathbf{f}^{g(\theta,\widehat{\boldsymbol{\epsilon}}_i)}(\mathbf{x})) + \lambda_1||\mathbf{M}_1||^2 + \lambda_2||\mathbf{M}_2||^2 + \lambda_3||\mathbf{b}||^2 \tag{3.11}$$

with $\widehat{\boldsymbol{\epsilon}}_i$ realisations of the random variable $\boldsymbol{\epsilon}$.

The derivative of this optimisation objective w.r.t. model parameters $\theta = \{\mathbf{M}_1, \mathbf{M}_2, \mathbf{b}\}$ is given by

$$\frac{\partial}{\partial\theta}\widehat{\mathcal{L}}_{\text{dropout}}(\theta) = -\frac{1}{M\tau}\sum_{i\in S}\frac{\partial}{\partial\theta}\log p(\mathbf{y}_i|\mathbf{f}^{g(\theta,\widehat{\boldsymbol{\epsilon}}_i)}(\mathbf{x})) + \frac{\partial}{\partial\theta}\left(\lambda_1||\mathbf{M}_1||^2 + \lambda_2||\mathbf{M}_2||^2 + \lambda_3||\mathbf{b}||^2\right).$$

The optimisation of a NN with dropout can then be seen as following algorithm 2.

There is a great sense of similarity between algorithm 1 for approximate inference in a Bayesian NN and algorithm 2 for dropout NN optimisation. Specifically, note the case of a Bayesian NN with approximating distribution $q(\boldsymbol{\omega})$ s.t. $\boldsymbol{\omega} = \{\text{diag}(\boldsymbol{\epsilon}_1)\mathbf{M}_1, \text{diag}(\boldsymbol{\epsilon}_2)\mathbf{M}_2, \mathbf{b}\}$ with $p(\boldsymbol{\epsilon}_l)$ $(l = 1, 2)$ a product of Bernoulli distributions with probability $1 - p_l$ (which we will refer to as a *Bernoulli variational distribution* or a *dropout variational distribution*). The only differences between algorithm 1 and algorithm 2 are

1. the regularisation term derivatives ($\text{KL}(q_\theta(\boldsymbol{\omega})||p(\boldsymbol{\omega}))$ in algo. 1 and $\lambda_1||\mathbf{M}_1||^2 + \lambda_2||\mathbf{M}_2||^2 + \lambda_3||\mathbf{b}||^2$ in algo. 2),
2. and the scale of $\widehat{\Delta\theta}$ (multiplied by a *constant* $\frac{1}{N\tau}$ in algo. 2).

Algorithm 2 Optimisation of a neural network with dropout

1: Given dataset $\mathbf{X}, \mathbf{Y}$,
2: Define learning rate schedule η,
3: Initialise parameters θ randomly.
4: **repeat**
5: Sample M random variables $\widehat{\boldsymbol{\epsilon}}_i \sim p(\boldsymbol{\epsilon})$, S a random subset of $\{1, .., N\}$ of size M.
6: Calculate derivative w.r.t. θ:

$$\widehat{\Delta\theta} \leftarrow -\frac{1}{M\tau}\sum_{i\in S}\frac{\partial}{\partial\theta}\log p(\mathbf{y}_i|\mathbf{f}^{g(\theta,\widehat{\boldsymbol{\epsilon}}_i)}(\mathbf{x})) + \frac{\partial}{\partial\theta}\left(\lambda_1||\mathbf{M}_1||^2 + \lambda_2||\mathbf{M}_2||^2 + \lambda_3||\mathbf{b}||^2\right).$$

7: Update θ:
$\theta \leftarrow \theta + \eta\widehat{\Delta\theta}$.
8: **until** θ has converged.

More specifically, if we define the prior $p(\boldsymbol{\omega})$ s.t. the following holds:

$$\frac{\partial}{\partial\theta}\mathrm{KL}(q_\theta(\boldsymbol{\omega})||p(\boldsymbol{\omega})) = \frac{\partial}{\partial\theta}N\tau\left(\lambda_1||\mathbf{M}_1||^2 + \lambda_2||\mathbf{M}_2||^2 + \lambda_3||\mathbf{b}||^2\right) \tag{3.12}$$

(referred to as the *KL condition*), we would have the following relation between the derivatives of objective (3.11) and objective (3.7):

$$\frac{\partial}{\partial\theta}\widehat{\mathcal{L}}_{\text{dropout}}(\theta) = \frac{1}{N\tau}\frac{\partial}{\partial\theta}\widehat{\mathcal{L}}_{\text{MC}}(\theta)$$

with identical optimisation procedures!

We found that for a specific choice for the approximating distribution $q_\theta(\boldsymbol{\omega})$, VI results in identical optimisation procedure to that of a dropout NN. I would stress that this means that optimising *any* neural network with dropout is equivalent to a form of approximate inference in a probabilistic interpretation of the model[5]. This means that the optimal weights found through the optimisation of a dropout NN (using algo. 2) are the same as the optimal variational parameters in a Bayesian NN with the same structure. Further, this means that a network already trained with dropout *is* a Bayesian NN, thus possesses all the properties a Bayesian NN possesses.

We have so far concentrated mostly on the dropout SRT. As to alternative SRTs, remember that an approximating distribution $q_\theta(\boldsymbol{\omega})$ is defined through its re-parametrisation $\boldsymbol{\omega} = g(\theta, \boldsymbol{\epsilon})$. Various SRTs can be recovered for different re-parametrisations. For exam-

[5]Note that to get well-calibrated uncertainty estimates we have to optimise the dropout probability p as well as θ, for example through grid-search over over validation log probability. This is discussed further in §4.3

ple, multiplicative Gaussian noise [Srivastava et al., 2014] can be recovered by setting $g(\theta, \boldsymbol{\epsilon}) = \{\text{diag}(\boldsymbol{\epsilon}_1)\mathbf{M}_1, \text{diag}(\boldsymbol{\epsilon}_2)\mathbf{M}_2, \mathbf{b}\}$ with $p(\boldsymbol{\epsilon}_l)$ (for $l = 1, 2$) a product of $\mathcal{N}(1, \alpha)$ with positive-valued α[6]. This can be efficiently implemented by multiplying a network's units by i.i.d. draws from a $\mathcal{N}(1, \alpha)$. On the other hand, setting $g(\theta, \boldsymbol{\epsilon}) = \{\mathbf{M}_1 \odot \boldsymbol{\epsilon}_1, \mathbf{M}_2 \odot \boldsymbol{\epsilon}_2, \mathbf{b}\}$ with $p(\boldsymbol{\epsilon}_l)$ a product of Bernoulli random variables for each weight scalar we recover dropConnect [Wan et al., 2013]. This can be efficiently implemented by multiplying a network's weight scalars by i.i.d. draws from a Bernoulli distribution. It is interesting to note that Graves [2011]'s fully factorised approximation can be recovered by setting $g(\theta, \boldsymbol{\epsilon}) = \{\mathbf{M}_1 + \boldsymbol{\epsilon}_1, \mathbf{M}_2 + \boldsymbol{\epsilon}_2, \mathbf{b}\}$ with $p(\boldsymbol{\epsilon}_l)$ a product of $\mathcal{N}(0, \alpha)$ for each weight scalar. This SRT is often referred to as *additive Gaussian noise*.

3.2.3 KL condition

For VI to result in an identical optimisation procedure to that of a dropout NN, the KL condition (eq. (3.12)) has to be satisfied. Under what constraints does the KL condition hold? This depends on the model specification (selection of prior $p(\boldsymbol{\omega})$) as well as choice of approximating distribution $q_\theta(\boldsymbol{\omega})$. For example, it can be shown that setting the model prior to $p(\boldsymbol{\omega}) = \prod_{i=1}^L p(\mathbf{W}_i) = \prod_{i=1}^L \mathcal{N}(0, \mathbf{I}/l_i^2)$, in other words independent normal priors over each weight, with *prior length-scale*[7]

$$l_i^2 = \frac{2N\tau\lambda_i}{1 - p_i} \tag{3.13}$$

we have

$$\frac{\partial}{\partial\theta}\text{KL}(q_\theta(\boldsymbol{\omega})||p(\boldsymbol{\omega})) \approx \frac{\partial}{\partial\theta} N\tau(\lambda_1||\mathbf{M}_1||^2 + \lambda_2||\mathbf{M}_2||^2 + \lambda_3||\mathbf{b}||^2)$$

for a large enough number of hidden units and a Bernoulli variational distribution. This is discussed further in appendix A. Alternatively, a discrete prior distribution

$$p(\mathbf{w}) \propto e^{-\frac{l^2}{2}\mathbf{w}^T\mathbf{w}}$$

[6] For the multiplicative Gaussian noise this result was also presented in [Kingma et al., 2015], which was done in parallel to this work.

[7] A note on *mean-squared-error* losses: the mean-squared-error loss can be seen as a scaling of the Euclidean loss (eq. (1.1)) by a factor of 2, which implies that the factor of 2 in the length-scale should be removed. The mean-squared-error loss is used in many modern deep learning packages instead (or as) the Euclidean loss.

defined over a *finite* space $\mathbf{w} \in X$ satisfies the KL condition (eq. (3.12)) exactly. This is discussed in more detail in §6.5. For multiplicative Gaussian noise, Kingma et al. [2015] have shown that an improper log-uniform prior distribution satisfies our KL condition.

Notation remark. In the rest of this work we will use $\mathbf{M}$ and $\mathbf{W}$ interchangeably, with the understanding that in deterministic NNs the random variable $\mathbf{W}$ will follow a delta distribution with mean parameter $\mathbf{M}$.

Remark (What is a prior length-scale l_i^2?)**.** It is interesting to explain why we consider the parameter l to be a prior *length-scale* in a Bayesian NN when setting a Gaussian prior $\mathcal{N}(0, I/l^2)$ over the weights. To see this, re-parametrise the input prior distribution as $\mathbf{W}_1' \sim \mathcal{N}(0, I)$, with $\mathbf{W}_1 = \mathbf{W}_1'/l$:

$$\widehat{\mathbf{y}} = \sigma\left(\mathbf{x}\frac{\mathbf{W}_1'}{l} + \mathbf{b}\right)\mathbf{W}_2 = \sigma\left(\frac{\mathbf{x}}{l}\mathbf{W}_1' + \mathbf{b}\right)\mathbf{W}_2$$

i.e. placing a prior distribution $\mathcal{N}(0, I/l^2)$ over $\mathbf{W}_1$ can be replaced by scaling the inputs by $1/l$ with a $\mathcal{N}(0, I)$ prior instead. For example, multiplying the inputs by 100 (making the function smoother) and placing a prior length-scale $l = 100$ would give identical model output to placing a prior length-scale $l = 1$ with the original inputs. This means that the length-scale's unit of measure is identical to the inputs' one.
What does the prior length-scale mean? To see this, consider a real valued function $f(x)$, periodic with period P, and consider its Fourier expansion with K terms:

$$f_K(x) := \frac{A_0}{2} + \sum_{k=1}^{K} A_k \cdot \sin\left(\tfrac{2\pi k}{P}x + \phi_k\right).$$

This can be seen as a single hidden layer neural network with a non-linearity $\sigma(\cdot) := \sin(\cdot)$, input weights given by the Fourier frequencies $\mathbf{W}_1 := [\frac{2\pi k}{P}]_{k=1}^K$ (which are fixed and not learnt), $\mathbf{b} := [\phi_k]_{k=1}^K$ is the bias term of the hidden layer, the Fourier coefficients are the output weights $\mathbf{W}_2 := [A_k]_{k=1}^K$, and $\frac{A_0}{2}$ is the output bias (which can be omitted for centred data). For simplicity we assume that $\mathbf{W}_1$ is composed of only the Fourier frequencies for which the Fourier coefficients are not zero. For example, $\mathbf{W}_1$ might be composed of high frequencies, low frequencies, or a combination of the two.

This view of single hidden layer neural networks gives us some insights into the role of the different quantities used in a neural network. For example, erratic functions have high frequencies, i.e. high magnitude input weights $\mathbf{W}_1$. On the other hand, smooth slow-varying functions are composed of low frequencies, and as a result the magnitude of $\mathbf{W}_1$ is small. The magnitude of $\mathbf{W}_2$ determines how much different frequencies will be used to compose the output function $f_K(x)$. High magnitude $\mathbf{W}_2$ results in a large magnitude for the function's outputs, whereas low $\mathbf{W}_2$ magnitude gives function outputs scaled down and closer to zero.

When we place a prior distribution over the input weights of a BNN, we can capture this characteristic. Having $\mathbf{W}_1 \sim \mathcal{N}(0, I/l^2)$ a priori with long length-scale l results in weights with low magnitude, and as a result slow-varying induced functions. On the other hand, placing a prior distribution with a short length-scale gives high magnitude weights, and as a result erratic functions with high frequencies. This will be demonstrated empirically in §4.1.

Given the intuition about weight magnitude above, equation (3.13) can be re-written to cast some light on the structure of the weight-decay in a neural network[a]:

$$\lambda_i = \frac{l_i^2(1-p_i)}{2N\tau}. \tag{3.14}$$

A short length-scale l_i (corresponding to high frequency data) with high precision τ (equivalently, small observation noise) results in a small weight-decay λ_i—encouraging the model to fit the data well but potentially generalising badly. A long length-scale with low precision results in a large weight-decay—and stronger regularisation over the weights. This trade-off between the length-scale and model precision results in different weight-decay values.

Lastly, I would comment on the choice of placing a distribution over the rows of a weight matrix rather than factorising it over each row's elements. Gal and Turner [2015] offered a derivation related to the Fourier expansion above, where a function drawn from a Gaussian process (GP) was approximated through a finite Fourier decomposition of the GP's covariance function. This derivation has many properties in common with the view above. Interestingly, in the multivariate $\mathbf{f}(\mathbf{x})$ case the Fourier frequencies are given in the columns of the equivalent weight matrix $\mathbf{W}_1$ of size Q (input dimension) by K (number of expansion terms). This generalises on the univariate case above where $\mathbf{W}_1$ is of dimensions $Q = 1$ by K and each entry (column) is a single frequency. Factorising the weight matrix approximating

distribution $q_\theta(\mathbf{W}_1)$ over its *rows* rather than columns captures correlations over the function's frequencies.

[a]Note that with a *mean-squared-error* loss the factor of 2 should be removed.

3.3 Model uncertainty in Bayesian neural networks

We next derive results extending on the above showing that model uncertainty can be obtained from NN models that make use of SRTs such as dropout.

Recall that our approximate predictive distribution is given by eq. (2.4):

$$q_\theta^*(\mathbf{y}^*|\mathbf{x}^*) = \int p(\mathbf{y}^*|\mathbf{f}^{\boldsymbol{\omega}}(\mathbf{x}^*))q_\theta^*(\boldsymbol{\omega})\mathrm{d}\boldsymbol{\omega} \tag{3.15}$$

where $\boldsymbol{\omega} = \{\mathbf{W}_i\}_{i=1}^L$ is our set of random variables for a model with L layers, $\mathbf{f}^{\boldsymbol{\omega}}(\mathbf{x}^*)$ is our model's stochastic output, and $q_\theta^*(\boldsymbol{\omega})$ is an optimum of eq. (3.7).

We will perform moment-matching and estimate the first two moments of the predictive distribution empirically. The first moment can be estimated as follows:

Proposition 2. *Given* $p(\mathbf{y}^*|\mathbf{f}^{\boldsymbol{\omega}}(\mathbf{x}^*)) = \mathcal{N}(\mathbf{y}^*; \mathbf{f}^{\boldsymbol{\omega}}(\mathbf{x}^*), \tau^{-1}\mathbf{I})$ *for some* $\tau > 0$, $\mathbb{E}_{q_\theta^*(\mathbf{y}^*|\mathbf{x}^*)}[\mathbf{y}^*]$ *can be estimated with the unbiased estimator*

$$\widetilde{\mathbb{E}}[\mathbf{y}^*] := \frac{1}{T}\sum_{t=1}^T \mathbf{f}^{\widehat{\boldsymbol{\omega}}_t}(\mathbf{x}^*) \xrightarrow[T\to\infty]{} \mathbb{E}_{q_\theta^*(\mathbf{y}^*|\mathbf{x}^*)}[\mathbf{y}^*] \tag{3.16}$$

with $\widehat{\boldsymbol{\omega}}_t \sim q_\theta^*(\boldsymbol{\omega})$.

Proof.

$$\begin{aligned}
\mathbb{E}_{q_\theta^*(\mathbf{y}^*|\mathbf{x}^*)}[\mathbf{y}^*] &= \int \mathbf{y}^* q_\theta^*(\mathbf{y}^*|\mathbf{x}^*)\mathrm{d}\mathbf{y}^* \\
&= \int\int \mathbf{y}^* \mathcal{N}(\mathbf{y}^*; \mathbf{f}^{\boldsymbol{\omega}}(\mathbf{x}^*), \tau^{-1}\mathbf{I}) q_\theta^*(\boldsymbol{\omega})\mathrm{d}\boldsymbol{\omega}\mathrm{d}\mathbf{y}^* \\
&= \int \left(\int \mathbf{y}^* \mathcal{N}(\mathbf{y}^*; \mathbf{f}^{\boldsymbol{\omega}}(\mathbf{x}^*), \tau^{-1}\mathbf{I})\mathrm{d}\mathbf{y}^*\right) q_\theta^*(\boldsymbol{\omega})\mathrm{d}\boldsymbol{\omega} \\
&= \int \mathbf{f}^{\boldsymbol{\omega}}(\mathbf{x}^*) q_\theta^*(\boldsymbol{\omega})\mathrm{d}\boldsymbol{\omega},
\end{aligned}$$

giving the unbiased estimator $\widetilde{\mathbb{E}}[\mathbf{y}^*] := \frac{1}{T}\sum_{t=1}^T \mathbf{f}^{\widehat{\boldsymbol{\omega}}_t}(\mathbf{x}^*)$ following MC integration with T samples. □

When used with dropout, we refer to this Monte Carlo estimate (3.16) as *MC dropout*. In practice MC dropout is equivalent to performing T stochastic forward passes through the network and averaging the results. For dropout, this result has been presented in the literature before as *model averaging* [Srivastava et al., 2014]. We have given a new derivation for this result which allows us to derive mathematically grounded uncertainty estimates as well, and generalises to all SRTs (including SRTs such as multiplicative Gaussian noise where the model averaging interpretation would result in infinitely many models). Srivastava et al. [2014, section 7.5] have reasoned based on empirical experimentation that the model averaging can be approximated by multiplying each network unit $\mathbf{h}_i$ by $1/(1-p_i)$ at test time, referred to as *standard dropout*. This can be seen as propagating the mean of each layer to the next. Below (in section §4.4) we give results showing that there exist models in which standard dropout gives a bad approximation to the model averaging.

We estimate the second raw moment (for regression) using the following proposition:

Proposition 3.
Given $p(\mathbf{y}^*|\mathbf{f}^{\boldsymbol{\omega}}(\mathbf{x}^*)) = \mathcal{N}(\mathbf{y}^*; \mathbf{f}^{\boldsymbol{\omega}}(\mathbf{x}^*), \tau^{-1}\mathbf{I})$ *for some* $\tau > 0$, $\mathbb{E}_{q_\theta^*(\mathbf{y}^*|\mathbf{x}^*)}\left[(\mathbf{y}^*)^T(\mathbf{y}^*)\right]$ *can be estimated with the unbiased estimator*

$$\widetilde{\mathbb{E}}\left[(\mathbf{y}^*)^T(\mathbf{y}^*)\right] := \tau^{-1}\mathbf{I} + \frac{1}{T}\sum_{t=1}^{T} \mathbf{f}^{\widehat{\boldsymbol{\omega}}_t}(\mathbf{x}^*)^T \mathbf{f}^{\widehat{\boldsymbol{\omega}}_t}(\mathbf{x}^*) \xrightarrow{T\to\infty} \mathbb{E}_{q_\theta^*(\mathbf{y}^*|\mathbf{x}^*)}\left[(\mathbf{y}^*)^T(\mathbf{y}^*)\right]$$

with $\widehat{\boldsymbol{\omega}}_t \sim q_\theta^*(\boldsymbol{\omega})$ *and* $\mathbf{y}^*, \mathbf{f}^{\widehat{\boldsymbol{\omega}}_t}(\mathbf{x}^*)$ *row vectors (thus the sum is over the outer-products).*

Proof.

$$\begin{aligned}
\mathbb{E}_{q_\theta^*(\mathbf{y}^*|\mathbf{x}^*)}\left[(\mathbf{y}^*)^T(\mathbf{y}^*)\right] &= \int \left(\int (\mathbf{y}^*)^T(\mathbf{y}^*) p(\mathbf{y}^*|\mathbf{x}^*, \boldsymbol{\omega}) \mathrm{d}\mathbf{y}^* \right) q_\theta^*(\boldsymbol{\omega}) \mathrm{d}\boldsymbol{\omega} \\
&= \int \left(\mathrm{Cov}_{p(\mathbf{y}^*|\mathbf{x}^*,\boldsymbol{\omega})}[\mathbf{y}^*] + \mathbb{E}_{p(\mathbf{y}^*|\mathbf{x}^*,\boldsymbol{\omega})}[\mathbf{y}^*]^T \mathbb{E}_{p(\mathbf{y}^*|\mathbf{x}^*,\boldsymbol{\omega})}[\mathbf{y}^*] \right) q_\theta^*(\boldsymbol{\omega}) \mathrm{d}\boldsymbol{\omega} \\
&= \int \left(\tau^{-1}\mathbf{I} + \mathbf{f}^{\boldsymbol{\omega}}(\mathbf{x}^*)^T \mathbf{f}^{\boldsymbol{\omega}}(\mathbf{x}^*) \right) q_\theta^*(\boldsymbol{\omega}) \mathrm{d}\boldsymbol{\omega}
\end{aligned}$$

giving the unbiased estimator $\widetilde{\mathbb{E}}\left[(\mathbf{y}^*)^T(\mathbf{y}^*)\right] := \tau^{-1}\mathbf{I} + \frac{1}{T}\sum_{t=1}^{T} \mathbf{f}^{\widehat{\boldsymbol{\omega}}_t}(\mathbf{x}^*)^T \mathbf{f}^{\widehat{\boldsymbol{\omega}}_t}(\mathbf{x}^*)$ following MC integration with T samples. □

To obtain the model's predictive variance we use the unbiased estimator:

$$\widetilde{\mathrm{Var}}[\mathbf{y}^*] := \tau^{-1}\mathbf{I} + \frac{1}{T}\sum_{t=1}^{T} \mathbf{f}^{\widehat{\boldsymbol{\omega}}_t}(\mathbf{x}^*)^T \mathbf{f}^{\widehat{\boldsymbol{\omega}}_t}(\mathbf{x}^*) - \widetilde{\mathbb{E}}[\mathbf{y}^*]^T \widetilde{\mathbb{E}}[\mathbf{y}^*]$$

$$\xrightarrow[T\to\infty]{} \mathrm{Var}_{q_\theta^*(\mathbf{y}^*|\mathbf{x}^*)}[\mathbf{y}^*]$$

which equals the sample variance of T stochastic forward passes through the NN plus the inverse model precision.

How can we find the model precision? In practice in the deep learning literature we often grid-search over the weight-decay λ to minimise validation error. Then, given a weight-decay λ_i (and prior length-scale l_i), eq. (3.13) can be re-written to find the model precision[8]:

$$\tau = \frac{(1-p)l_i^2}{2N\lambda_i}. \tag{3.17}$$

Remark (Predictive variance and posterior variance)**.** It is important to note the difference between the variance of the approximating distribution $q_\theta(\boldsymbol{\omega})$ and the variance of the predictive distribution $q_\theta(\mathbf{y}|\mathbf{x})$ (eq. (3.15)).
To see this, consider the illustrative example of an approximating distribution with fixed mean and variance used with the first weight layer $\mathbf{W}_1$, for example a standard Gaussian $\mathcal{N}(0, I)$. Further, assume for the sake of argument that delta distributions (or Gaussians with very small variances) are used to approximate the posterior over the layers following the first layer. Given enough follow-up layers we can capture any function to arbitrary precision—including the inverse cumulative distribution function (CDF) of any distribution (similarly to the remark in §2.2.1, but with the addition of a Gaussian CDF as we don't have a uniform on the first layer in this case). Passing the distribution from the first layer through the rest of the layers transforms the standard Gaussian with this inverse CDF, resulting in any arbitrary distribution as determined by the CDF.
In this example, even though the variance of each weight layer is constant, the variance of the predictive distribution can take any value depending on the learnt CDF. This example can be extended from a standard Gaussian approximating distribution to a mixture of Gaussians with fixed standard deviations, and to discrete distributions with fixed probability vectors (such as the dropout approximating distribution). Of course, in real world cases we would prefer to avoid modelling the

[8]Prior length-scale l_i can be fixed based on the density of the input data $\mathbf{X}$ and our prior belief as to the function's wiggliness, or optimised over as well (w.r.t. predictive log-likelihood over a validation set). The dropout probability is optimised using grid search similarly.

deep layers with delta approximating distributions since that would sacrifice our ability to capture model uncertainty.

Given a dataset $\mathbf{X}, \mathbf{Y}$ and a new data point $\mathbf{x}^*$ we can calculate the probability of possible output values $\mathbf{y}^*$ using the predictive probability $p(\mathbf{y}^*|\mathbf{x}^*, \mathbf{X}, \mathbf{Y})$. The log of the predictive likelihood captures how well the model fits the data, with larger values indicating better model fit. Our predictive log-likelihood (also referred to as *test log-likelihood*) can be approximated by MC integration of eq. (3.15) with T terms:

$$\begin{aligned}\widetilde{\log} p(\mathbf{y}^*|\mathbf{x}^*, \mathbf{X}, \mathbf{Y}) := \log\left(\frac{1}{T}\sum_{t=1}^{T} p(\mathbf{y}^*|\mathbf{x}^*, \boldsymbol{\omega}_t)\right) &\xrightarrow[T\to\infty]{} \log \int p(\mathbf{y}^*|\mathbf{x}^*, \boldsymbol{\omega}) q_\theta^*(\boldsymbol{\omega}) \mathrm{d}\boldsymbol{\omega} \\ &\approx \log \int p(\mathbf{y}^*|\mathbf{x}^*, \boldsymbol{\omega}) p(\boldsymbol{\omega}|\mathbf{X}, \mathbf{Y}) \mathrm{d}\boldsymbol{\omega} \\ &= \log p(\mathbf{y}^*|\mathbf{x}^*, \mathbf{X}, \mathbf{Y})\end{aligned}$$

with $\boldsymbol{\omega}_t \sim q_\theta^*(\boldsymbol{\omega})$ and since $q_\theta^*(\boldsymbol{\omega})$ is the minimiser of eq. (2.3). Note that this is a biased estimator since the expected quantity is transformed with the non-linear logarithm function, but the bias decreases as T increases.

For regression we can rewrite this last equation in a more numerically stable way[9]:

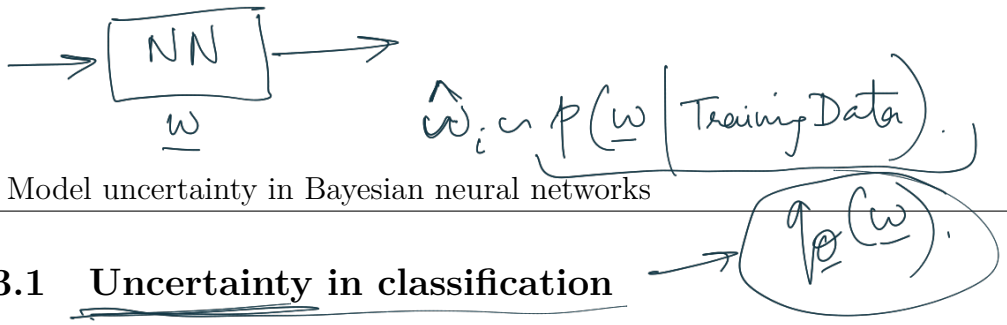
$$\widetilde{\log} p(\mathbf{y}^*|\mathbf{x}^*, \mathbf{X}, \mathbf{Y}) = \mathrm{logsumexp}\left(-\frac{1}{2}\tau||\mathbf{y} - \mathbf{f}^{\widehat{\omega}_t}(\mathbf{x}^*)||^2\right) - \log T - \frac{1}{2}\log 2\pi + \frac{1}{2}\log\tau \tag{3.18}$$

with our precision parameter τ.

Uncertainty quality can be determined from this quantity as well. Excessive uncertainty (large observation noise, or equivalently small model precision τ) results in a large penalty from the last term in the predictive log-likelihood. On the other hand, an over-confident model with large model precision compared to poor mean estimation results in a penalty from the first term—the distance $||\mathbf{y} - \mathbf{f}^{\widehat{\omega}_t}(\mathbf{x}^*)||^2$ gets amplified by τ which drives the exponent to zero.

Note that the normal NN model itself is not changed. To estimate the predictive mean and predictive uncertainty we simply collect the results of stochastic forward passes through the model. As a result, this information can be used with existing NN models trained with SRTs. Furthermore, the forward passes can be done concurrently, resulting in constant running time identical to that of standard NNs.

[9] logsumexp is the log-sum-exp function.



3.3.1 Uncertainty in classification

In regression we summarised predictive uncertainty by looking at the sample variance of multiple stochastic forward passes. In the classification setting, however, we need to rely on alternative approaches to summarise uncertainty[10]. We will analyse three approaches to summarise uncertainty within classification: variation ratios [Freeman, 1965], predictive entropy [Shannon, 1948], and mutual information [Shannon, 1948]. These measures capture different notions of uncertainty: model uncertainty and predictive uncertainty, and will be explained below. But first, how do we calculate these quantities in our setting?

To use variation ratios we would sample a label from the softmax probabilities at the end of each *stochastic forward pass* for a test input $\mathbf{x}$. Collecting a set of T labels y_t from multiple stochastic forward passes on the same input we can find the mode of the distribution[11] $c^* = \underset{c=1,\dots,C}{\arg\max} \sum_t \mathbb{1}[y^t = c]$, and the number of times it was sampled $f_{\mathbf{x}} = \sum_t \mathbb{1}[y^t = c^*]$. We then set

$$\text{variation-ratio}[\mathbf{x}] := 1 - \frac{f_{\mathbf{x}}}{T}. \tag{3.19}$$

The variation ratio is a measure of dispersion—how "spread" the distribution is around the mode. In the binary case, the variation ratio attains its maximum of 0.5 when the two classes are sampled equally likely, and its minimum of 0 when only a single class is sampled.

Remark (Variation ratios and approximate inference)**.** The variation ratio as it was formulated in [Freeman, 1965] and used above can be seen as approximating the quantity

$$1 - p(y = c^*|\mathbf{x}, \mathcal{D}_{\text{train}})$$

with $c^* = \underset{c=1,\dots,C}{\arg\max}\, p(y = c|\mathbf{x}, \mathcal{D}_{\text{train}})$. This is because for y^t the t'th class sampled for input $\mathbf{x}$ we have:

$$\frac{f_{\mathbf{x}}}{T} = \frac{1}{T}\sum_t \mathbb{1}[y^t = c^*] \xrightarrow[T\to\infty]{} \mathbb{E}_{q_\theta^*(y|\mathbf{x})}\Big[\mathbb{1}[y = c^*]\Big]$$

[10] These approaches are necessary since the probability vector resulting from a *deterministic* forward pass through the model does not capture confidence, as explained in figure 1.3.

[11] Here $\mathbb{1}[\cdot]$ is the indicator function.

$$
\begin{aligned}
&= q_\theta^*(y = c^*|\mathbf{x}) \\
&\approx p(y = c^*|\mathbf{x}, \mathcal{D}_{\text{train}})
\end{aligned}
$$

and,

$$
\begin{aligned}
c^* = \underset{c=1,\dots,C}{\arg\max} \sum_t \mathbb{1}[y^t = c] &= \underset{c=1,\dots,C}{\arg\max} \frac{1}{T} \sum_t \mathbb{1}[y^t = c] \\
&\xrightarrow[T\to\infty]{} \underset{c=1,\dots,C}{\arg\max} \mathbb{E}_{q_\theta^*(y|\mathbf{x})}[\mathbb{1}[y = c]] \\
&= \underset{c=1,\dots,C}{\arg\max} q_\theta^*(y = c|\mathbf{x}). \\
&\approx \underset{c=1,\dots,C}{\arg\max} p(y = c|\mathbf{x}, \mathcal{D}_{\text{train}})
\end{aligned}
$$

since $q_\theta^*(\boldsymbol{\omega})$ is a minimiser of the KL divergence to $p(\boldsymbol{\omega}|\mathcal{D}_{\text{train}})$ and therefore $q_\theta^*(y|\mathbf{x}) \approx \int p(y|\mathbf{f}^\omega(\mathbf{x}))p(\boldsymbol{\omega}|\mathcal{D}_{\text{train}})\mathrm{d}\boldsymbol{\omega}$ (following eq. (3.15)).

Unlike variation ratios, predictive entropy has its foundations in information theory. This quantity captures the average amount of information contained in the predictive distribution:

$$
\mathbb{H}[y|\mathbf{x}, \mathcal{D}_{\text{train}}] := -\sum_c p(y = c|\mathbf{x}, \mathcal{D}_{\text{train}}) \log p(y = c|\mathbf{x}, \mathcal{D}_{\text{train}}) \tag{3.20}
$$

summing over all possible classes c that y can take. Given a test point $\mathbf{x}$, the predictive entropy attains its maximum value when all classes are predicted to have equal uniform probability, and its minimum value of zero when one class has probability 1 and all others probability 0 (i.e. the prediction is certain).

In our setting, the predictive entropy can be approximated by collecting the probability vectors from T stochastic forward passes through the network, and for each class c averaging the probabilities of the class from each of the T probability vectors, replacing $p(y = c|\mathbf{x}, \mathcal{D}_{\text{train}})$ in eq. (3.20). In other words, we replace $p(y = c|\mathbf{x}, \mathcal{D}_{\text{train}})$ with $\frac{1}{T}\sum_t p(y = c|\mathbf{x}, \widehat{\boldsymbol{\omega}}_t)$, where $p(y = c|\mathbf{x}, \widehat{\boldsymbol{\omega}}_t)$ is the probability of input $\mathbf{x}$ to take class c with model parameters $\widehat{\boldsymbol{\omega}}_t \sim q_\theta^*(\boldsymbol{\omega})$:

$$
[p(y = 1|\mathbf{x}, \widehat{\boldsymbol{\omega}}_t), ..., p(y = C|\mathbf{x}, \widehat{\boldsymbol{\omega}}_t)] := \text{Softmax}(\mathbf{f}^{\widehat{\boldsymbol{\omega}}_t}(\mathbf{x})).
$$

Then,

$$\begin{aligned}
\tilde{\mathbb{H}}[y|\mathbf{x},\mathcal{D}_{\text{train}}] &:= -\sum_c \left(\frac{1}{T}\sum_t p(y=c|\mathbf{x},\widehat{\boldsymbol{\omega}}_t)\right)\log\left(\frac{1}{T}\sum_t p(y=c|\mathbf{x},\widehat{\boldsymbol{\omega}}_t)\right)\\
&\xrightarrow[T\to\infty]{} -\sum_c \left(\int p(y=c|\mathbf{x},\boldsymbol{\omega})q_\theta^*(\boldsymbol{\omega})\mathrm{d}\boldsymbol{\omega}\right)\log\left(\int p(y=c|\mathbf{x},\boldsymbol{\omega})q_\theta^*(\boldsymbol{\omega})\mathrm{d}\boldsymbol{\omega}\right)\\
&\approx -\sum_c \left(\int p(y=c|\mathbf{x},\boldsymbol{\omega})p(\boldsymbol{\omega}|\mathcal{D}_{\text{train}})\mathrm{d}\boldsymbol{\omega}\right)\log\left(\int p(y=c|\mathbf{x},\boldsymbol{\omega})p(\boldsymbol{\omega}|\mathcal{D}_{\text{train}})\mathrm{d}\boldsymbol{\omega}\right)\\
&= -\sum_c p(y=c|\mathbf{x},\mathcal{D}_{\text{train}})\log p(y=c|\mathbf{x},\mathcal{D}_{\text{train}})\\
&= \mathbb{H}[y|\mathbf{x},\mathcal{D}_{\text{train}}]
\end{aligned}$$

with $\widehat{\boldsymbol{\omega}}_t \sim q_\theta^*(\boldsymbol{\omega})$ and since $q_\theta^*(\boldsymbol{\omega})$ is the optimum of eq. (3.7). Note that this is a biased estimator since the unbiased estimator $\frac{1}{T}\sum_t p(y=c|\mathbf{x},\widehat{\boldsymbol{\omega}}_t) \xrightarrow[T\to\infty]{} \int p(y=c|\mathbf{x},\boldsymbol{\omega})q_\theta^*(\boldsymbol{\omega})\mathrm{d}\boldsymbol{\omega}$ is transformed through the non-linear function $\mathbb{H}[\cdot]$. The bias of this estimator will decrease as T increases.

As an alternative to the predictive entropy, the mutual information between the prediction y and the posterior over the model parameters $\boldsymbol{\omega}$ offers a different measure of uncertainty:

$$\begin{aligned}
\mathbb{I}[y,\boldsymbol{\omega}|\mathbf{x},\mathcal{D}_{\text{train}}] &:= \mathbb{H}[y|\mathbf{x},\mathcal{D}_{\text{train}}] - \mathbb{E}_{p(\boldsymbol{\omega}|\mathcal{D}_{\text{train}})}\Big[\mathbb{H}[y|\mathbf{x},\boldsymbol{\omega}]\Big]\\
&= -\sum_c p(y=c|\mathbf{x},\mathcal{D}_{\text{train}})\log p(y=c|\mathbf{x},\mathcal{D}_{\text{train}})\\
&\quad + \mathbb{E}_{p(\boldsymbol{\omega}|\mathcal{D}_{\text{train}})}\Big[\sum_c p(y=c|\mathbf{x},\boldsymbol{\omega})\log p(y=c|\mathbf{x},\boldsymbol{\omega})\Big]
\end{aligned}$$

with c the possible classes y can take. This tractable view of the mutual information was suggested in [Houlsby et al., 2011] in the context of active learning. Test points $\mathbf{x}$ that maximise the mutual information are points on which the model is uncertain on average, yet there exist model parameters that erroneously produce predictions with high confidence.

The mutual information can be approximated in our setting in a similar way to the predictive entropy approximation:

$$\begin{aligned}
\tilde{\mathbb{I}}[y,\boldsymbol{\omega}|\mathbf{x},\mathcal{D}_{\text{train}}] &:= -\sum_c \left(\frac{1}{T}\sum_t p(y=c|\mathbf{x},\widehat{\boldsymbol{\omega}}_t)\right)\log\left(\frac{1}{T}\sum_t p(y=c|\mathbf{x},\widehat{\boldsymbol{\omega}}_t)\right)\\
&\quad + \frac{1}{T}\sum_{c,t} p(y=c|\mathbf{x},\widehat{\boldsymbol{\omega}}_t)\log p(y=c|\mathbf{x},\widehat{\boldsymbol{\omega}}_t)\\
&\xrightarrow[T\to\infty]{} \mathbb{H}[y|\mathbf{x},\mathcal{D}_{\text{train}}] - \mathbb{E}_{q_\theta^*(\boldsymbol{\omega})}\Big[\mathbb{H}[y|\mathbf{x},\boldsymbol{\omega}]\Big]
\end{aligned}$$

$$\approx \mathbb{I}[y, \boldsymbol{\omega} | \mathbf{x}, \mathcal{D}_{\text{train}}]$$

with $\hat{\boldsymbol{\omega}}_t \sim q_\theta^*(\boldsymbol{\omega})$.

Some intuition. To understand the different measures for uncertainty, we shall look at three concrete examples in binary classification of dogs and cats given an input image. More specifically, we will look at the sets of probability vectors obtained from multiple stochastic forward passes, and the uncertainty measures resulting from these sets. The three examples are where the probabilities for the class "dog" in all vectors are

1. all equal to 1 (i.e. the probability vectors collected are $\{(1, 0), ..., (1, 0)\}$),
2. all equal to 0.5 (i.e. the probability vectors collected are $\{(0.5, 0.5), ..., (0.5, 0.5)\}$), and
3. half of the probabilities sampled equal to 0 and half of the probabilities equal to 1 (i.e. the probability vectors collected are $\{(1, 0), (0, 1), (0, 1), ..., (1, 0)\}$ for example).

In example (1) the prediction has high confidence, whereas in examples (2) and (3) the prediction has low confidence. These are examples of *predictive* uncertainty. Compared to this notion of confidence, in examples (1) and (2) the *model* is confident about its output since it gives identical probabilities in multiple forward passes. On the other hand, in the last example (3) the model is uncertain about its output, corresponding to the case in figure 1.3 (where the layer before the softmax has high uncertainty). This is an example of *model uncertainty*.

In example (1), both variation ratios, predictive entropy, and the mutual information would return value 0, all measures indicating high *confidence*. In example (3) variation ratios, predictive entropy, and the mutual information would all return value 0.5, all measures indicating high *uncertainty*. All three measures of uncertainty agree on these two examples.

However, in example (2) variation ratios and predictive entropy would return value 0.5, whereas the mutual information would return value 0. In this case variation ratios and predictive entropy capture the *uncertainty in the prediction*, whereas the mutual information captures the *model's confidence* in its output. This information can be used for example in *active learning*, and will be demonstrated in section §5.2.

3.3.2 Difficulties with the approach

Our technique is simple: perform several stochastic forward passes through the model, and look at the sample mean and variance. But it has several shortcomings worth discussing.

First, even though the training time of our model is identical to that of existing models in the field, the test time is scaled by T—the number of averaged forward passes through the network. This may not be of real concern in some real world applications, as NNs are often implemented on distributed hardware. Distributed hardware allows us to obtain MC estimates in constant time almost trivially, by transferring an input to a GPU and setting a mini-batch composed of the same input multiple times. In dropout for example we sample different Bernoulli realisations for each output unit and each mini-batch input, which results in a matrix of probabilities. Each row in the matrix is the output of the dropout network on the same input generated with different random variable realisations (dropout masks). Averaging over the rows results in the MC dropout estimate.

Another concern is that the model's uncertainty is not calibrated. A calibrated model is one in which the predictive probabilities match the empirical frequency of the data. The lack of calibration can be seen through the derivation's relation to Gaussian processes [Gal and Ghahramani, 2015b]. Gaussian processes' uncertainty is known to not be calibrated—the Gaussian process's uncertainty depends on the covariance function chosen, which is shown in [Gal and Ghahramani, 2015b] to be equivalent to the non-linearities and prior over the weights. The choice of a GP's covariance function follows from our assumptions about the data. If we believe, for example, that the model's uncertainty should increase far from the data we might choose the squared exponential covariance function.

For many practical applications the lack of calibration means that model uncertainty can increase for large magnitude data points or be of different scale for different datasets. To calibrate model uncertainty in regression tasks we can scale the uncertainty linearly to remove data magnitude effects, and manipulate uncertainty percentiles to compare among different datasets. This can be done by finding the number of validation set points having larger uncertainty than that of a test point. For example, if a test point has predictive standard deviation 5, whereas almost all validation points have standard deviation ranging from 0.2 to 2, then the test point's uncertainty value will be in the top percentile of the validation set uncertainty measures, and the model will be considered as very uncertain about the test point compared to the validation data. However, another model might give the same test point predictive standard deviation of 5 with most of the validation data given predictive standard deviation ranging from 10 to 15. In this model the test point's uncertainty measure will be in the lowest percentile of validation set uncertainty measures, and the model will be considered as fairly confident about the test point with respect to the validation data.

One last concern is a known limitation of VI. Variational inference is known to underestimates predictive variance [Turner and Sahani, 2011], a property descendent from our objective (which penalises $q_\theta(\boldsymbol{\omega})$ for placing mass where $p(\boldsymbol{\omega}|\mathbf{X}, \mathbf{Y})$ has no mass, but less so for not placing mass where it should). Several solutions exist for this (such as [Giordano et al., 2015]), with different limitations for their practicality. Uncertainty under-estimation does not seem to be of real concern in practice though, and the variational approximation seems to work well in practical applications as we will see in the next chapter.

3.4 Approximate inference in complex models

We finish the chapter by extending the approximate inference technique above to more complex models such as convolutional neural networks and recurrent neural networks. This will allow us to obtain model uncertainty for models defined over sequence based datasets or for image data. We describe the approximate inference using Bernoulli variational distributions for convenience of presentation, although any SRT could be used instead.

3.4.1 Bayesian convolutional neural networks

In existing convolutional neural networks (CNNs) literature dropout is mostly used after inner-product layers at the end of the model alone. This can be seen as applying a finite deterministic transformation to the data before feeding it into a Bayesian NN. As such, model uncertainty can still be obtained for such models, but an interesting question is whether we could use approximate Bayesian inference over the full CNN. Here we wish to integrate over the convolution layers (kernels) of the CNN as well. To implement a Bayesian CNN we could apply dropout after all convolution layers as well as inner-product layers, and evaluate the model's predictive posterior using eq. (3.8) at test time. Note though that generalisations to SRTs other than dropout are also possible and easy to implement.

Recall the structure of a CNN described in §1.1 and in figure 1.1. In more detail, the input to the i'th convolution layer is represented as a 3 dimensional tensor $\mathbf{x} \in \mathbb{R}^{H_{i-1} \times W_{i-1} \times K_{i-1}}$ with height H_{i-1}, width W_{i-1}, and K_{i-1} channels. A convolution layer is then composed of a sequence of K_i kernels (weight tensors): $\mathbf{k}_k \in \mathbb{R}^{h \times w \times K_{i-1}}$ for $k = 1, ..., K_i$. Here we assume kernel height h, kernel width w, and the last dimension to match the number of channels in the input layer: K_{i-1}. Convolving

the kernels with the input (with a given stride s) then results in an output layer of dimensions $\mathbf{y} \in \mathbb{R}^{H'_{i-1} \times W'_{i-1} \times K_i}$ with H'_{i-1} and W'_{i-1} being the new height and width, and K_i channels—the number of kernels. Each element $y_{i,j,k}$ is the sum of the element-wise product of kernel $\mathbf{k}_k$ with a corresponding patch in the input image $\mathbf{x}$: $[[x_{i-h/2,j-w/2,1}, ..., x_{i+h/2,j+w/2,1}], ..., [x_{i-h/2,j-w/2,K_{i-1}}, ..., x_{i+h/2,j+w/2,K_{i-1}}]]$.

To integrate over the kernels, we reformulate the convolution as a linear operation. Let $\mathbf{k}_k \in \mathbb{R}^{h \times w \times K_{i-1}}$ for $k = 1, ..., K_i$ be the CNN's kernels with height h, width w, and K_{i-1} channels in the i'th layer. The input to the layer is represented as a 3 dimensional tensor $\mathbf{x} \in \mathbb{R}^{H_{i-1} \times W_{i-1} \times K_{i-1}}$ with height H_{i-1}, width W_{i-1}, and K_{i-1} channels. Convolving the kernels with the input with a given stride s is equivalent to extracting patches from the input and performing a matrix product: we extract $h \times w \times K_{i-1}$ dimensional patches from the input with stride s and vectorise these. Collecting the vectors in the rows of a matrix we obtain a new representation for our input $\overline{\mathbf{x}} \in \mathbb{R}^{n \times hwK_{i-1}}$ with n patches. The vectorised kernels form the columns of the weight matrix $\mathbf{W}_i \in \mathbb{R}^{hwK_{i-1} \times K_i}$. The convolution operation is then equivalent to the matrix product $\overline{\mathbf{x}}\mathbf{W}_i \in \mathbb{R}^{n \times K_i}$. The columns of the output can be re-arranged into a 3 dimensional tensor $\mathbf{y} \in \mathbb{R}^{H_i \times W_i \times K_i}$ (since $n = H_i \times W_i$). Pooling can then be seen as a non-linear operation on the matrix $\mathbf{y}$. Note that the pooling operation is a non-linearity applied after the linear convolution counterpart to ReLU or Tanh non-linearities.

To make the CNN into a probabilistic model we place a prior distribution over each kernel and approximately integrate each kernels-patch pair with Bernoulli variational distributions. We sample Bernoulli random variables $\boldsymbol{\epsilon}_{i,j,n}$ and multiply patch n by the weight matrix $\mathbf{W}_i \cdot \text{diag}([\boldsymbol{\epsilon}_{i,j,n}]_{j=1}^{K_i})$. This product is equivalent to an approximating distribution modelling each kernel-patch pair with a distinct random variable, tying the means of the random variables over the patches. The distribution randomly sets kernels to zero for different patches. This approximating distribution is also equivalent to applying dropout for each element in the tensor $\mathbf{y}$ before pooling. Implementing our Bayesian CNN is therefore as simple as using dropout after every convolution layer before pooling.

The standard dropout test time approximation (scaling hidden units by $1/(1 - p_i)$) does not perform well when dropout is applied after convolutions—this is a negative result we identified empirically. We solve this by approximating the predictive distribution following eq. (3.8), averaging stochastic forward passes through the model at test time (using MC dropout). We assess the model above with an extensive set of experiments studying its properties in §4.4.

3.4.2 Bayesian recurrent neural networks

We next develop inference with Bernoulli variational distributions for recurrent neural networks (RNNs), although generalisations to SRTs other than dropout are trivial. We will concentrate on simple RNN models for brevity of notation. Derivations for LSTM and GRU follow similarly. Given input sequence $\mathbf{x} = [\mathbf{x}_1, ..., \mathbf{x}_T]$ of length T, a simple RNN is formed by a repeated application of a deterministic function $\mathbf{f_h}$. This generates a hidden state $\mathbf{h}_t$ for time step t:

$$\mathbf{h}_t = \mathbf{f_h}(\mathbf{x}_t, \mathbf{h}_{t-1}) = \sigma(\mathbf{x}_t \mathbf{W_h} + \mathbf{h}_{t-1} \mathbf{U_h} + \mathbf{b_h})$$

for some non-linearity σ. The model output can be defined, for example, as

$$\mathbf{f_y}(\mathbf{h}_T) = \mathbf{h}_T \mathbf{W_y} + \mathbf{b_y}.$$

To view this RNN as a probabilistic model we regard $\boldsymbol{\omega} = \{\mathbf{W_h}, \mathbf{U_h}, \mathbf{b_h}, \mathbf{W_y}, \mathbf{b_y}\}$ as random variables (following Gaussian prior distributions). To make the dependence on $\boldsymbol{\omega}$ clear, we write $\mathbf{f}_\mathbf{y}^\boldsymbol{\omega}$ for $\mathbf{f_y}$ and similarly for $\mathbf{f}_\mathbf{h}^\boldsymbol{\omega}$. We define our probabilistic model's likelihood as above (section 2.1). The posterior over random variables $\boldsymbol{\omega}$ is rather complex, and we approximate it with a variational distribution $q(\boldsymbol{\omega})$. For the dropout SRT for example we may use a Bernoulli approximating distribution.

Recall our VI optimisation objective eq. (3.1). Evaluating each log likelihood term in eq. (3.1) with our RNN model we have

$$\begin{aligned}\int q(\boldsymbol{\omega}) \log p(\mathbf{y}|\mathbf{f}_\mathbf{y}^\boldsymbol{\omega}(\mathbf{h}_T)) \mathrm{d}\boldsymbol{\omega} &= \int q(\boldsymbol{\omega}) \log p\left(\mathbf{y} \middle| \mathbf{f}_\mathbf{y}^\boldsymbol{\omega}\left(\mathbf{f}_\mathbf{h}^\boldsymbol{\omega}(\mathbf{x}_T, \mathbf{h}_{T-1})\right)\right) \mathrm{d}\boldsymbol{\omega} \\ &= \int q(\boldsymbol{\omega}) \log p\left(\mathbf{y} \middle| \mathbf{f}_\mathbf{y}^\boldsymbol{\omega}\left(\mathbf{f}_\mathbf{h}^\boldsymbol{\omega}(\mathbf{x}_T, \mathbf{f}_\mathbf{h}^\boldsymbol{\omega}(...\mathbf{f}_\mathbf{h}^\boldsymbol{\omega}(\mathbf{x}_1, \mathbf{h}_0)...))\right)\right) \mathrm{d}\boldsymbol{\omega}\end{aligned}$$

with $\mathbf{h}_0 = \mathbf{0}$. We approximate this with MC integration with a single sample:

$$\approx \log p\left(\mathbf{y} \middle| \mathbf{f}_\mathbf{y}^{\widehat{\boldsymbol{\omega}}}\left(\mathbf{f}_\mathbf{h}^{\widehat{\boldsymbol{\omega}}}(\mathbf{x}_T, \mathbf{f}_\mathbf{h}^{\widehat{\boldsymbol{\omega}}}(...\mathbf{f}_\mathbf{h}^{\widehat{\boldsymbol{\omega}}}(\mathbf{x}_1, \mathbf{h}_0)...))\right)\right),$$

with $\widehat{\boldsymbol{\omega}} \sim q(\boldsymbol{\omega})$, resulting in an unbiased estimator to each sum term[12].

[12]Note that for brevity we did not re-parametrise the integral, although this should be done to obtain low variance derivative estimators.

This estimator is plugged into equation (3.1) to obtain our minimisation objective

$$\widehat{\mathcal{L}}_{\text{MC}} = -\sum_{i=1}^{N} \log p\left(\mathbf{y}_i \middle| \mathbf{f}_{\mathbf{y}}^{\widehat{\boldsymbol{\omega}}_i}\left(\mathbf{f}_{\mathbf{h}}^{\widehat{\boldsymbol{\omega}}_i}(\mathbf{x}_{i,T}, \mathbf{f}_{\mathbf{h}}^{\widehat{\boldsymbol{\omega}}_i}(...\mathbf{f}_{\mathbf{h}}^{\widehat{\boldsymbol{\omega}}_i}(\mathbf{x}_{i,1}, \mathbf{h}_0)...))\right)\right) + \text{KL}(q(\boldsymbol{\omega})||p(\boldsymbol{\omega})). \quad (3.21)$$

Note that for each sequence $\mathbf{x}_i$ we sample a new realisation $\widehat{\boldsymbol{\omega}}_i = \{\widehat{\mathbf{W}}_{\mathbf{h}}^i, \widehat{\mathbf{U}}_{\mathbf{h}}^i, \widehat{\mathbf{b}}_{\mathbf{h}}^i, \widehat{\mathbf{W}}_{\mathbf{y}}^i, \widehat{\mathbf{b}}_{\mathbf{y}}^i\}$, and that each symbol in the sequence $\mathbf{x}_i = [\mathbf{x}_{i,1}, ..., \mathbf{x}_{i,T}]$ is passed through the function $\mathbf{f}_{\mathbf{h}}^{\widehat{\boldsymbol{\omega}}_i}$ with *the same weight realisations* $\widehat{\mathbf{W}}_{\mathbf{h}}^i, \widehat{\mathbf{U}}_{\mathbf{h}}^i, \widehat{\mathbf{b}}_{\mathbf{h}}^i$ *used at every time step* $t \leq T$.

In the dropout case, evaluating the model output $\mathbf{f}_{\mathbf{y}}^{\widehat{\boldsymbol{\omega}}}(\cdot)$ with sample $\widehat{\boldsymbol{\omega}}$ corresponds to randomly zeroing (masking) rows in each weight matrix $\mathbf{W}$ during the forward pass. In our RNN setting with a sequence input, each weight matrix row is randomly masked once, and importantly the same mask is used through all time steps. When viewed as a stochastic regularisation technique, our induced dropout variant is therefore identical to implementing dropout in RNNs with the *same network units dropped at each time step*, randomly dropping inputs, outputs, and recurrent connections. Predictions can be approximated by either propagating the mean of each layer to the next (referred to as the *standard dropout approximation*), or by performing dropout at test time and averaging results (MC dropout, eq. (3.16)).

Remark (Bayesian versus ensembling interpretation of dropout)**.** Apart from our Bayesian approximation interpretation, dropout in deep networks can also be seen as following an ensembling interpretation [Srivastava et al., 2014]. This interpretation also leads to MC dropout at test time. But the ensembling interpretation does not determine whether the ensemble should be over the network units or the weights. For example, in an RNN this view will *not* lead to our dropout variant, unless the ensemble is *defined to tie the weights of the network* ad hoc. This is in comparison to the Bayesian approximation view where the weight tying is forced by the probabilistic interpretation of the model.
The same can be said about the latent variable model view of dropout [Maeda, 2014] where a constraint over the weights would have to be added ad hoc to derive the results presented here.

Certain RNN models such as LSTMs [Graves et al., 2013; Hochreiter and Schmidhuber, 1997] and GRUs [Cho et al., 2014] use different *gates* within the RNN units. For example, an LSTM is defined by setting four gates: "input", "forget", "output", and an "input modulation gate",

$$\underline{\mathbf{i}} = \text{sigm}\left(\mathbf{h}_{t-1}\mathbf{U}_i + \mathbf{x}_t\mathbf{W}_i\right) \qquad \underline{\mathbf{f}} = \text{sigm}\left(\mathbf{h}_{t-1}\mathbf{U}_f + \mathbf{x}_t\mathbf{W}_f\right)$$

$$\underline{\mathbf{o}} = \text{sigm}\Big(\mathbf{h}_{t-1}\mathbf{U}_o + \mathbf{x}_t\mathbf{W}_o\Big) \qquad \underline{\mathbf{g}} = \tanh\Big(\mathbf{h}_{t-1}\mathbf{U}_g + \mathbf{x}_t\mathbf{W}_g\Big)$$
$$\mathbf{c}_t = \underline{\mathbf{f}} \odot \mathbf{c}_{t-1} + \underline{\mathbf{i}} \odot \underline{\mathbf{g}} \qquad \mathbf{h}_t = \underline{\mathbf{o}} \odot \tanh(\mathbf{c}_t) \tag{3.22}$$

with $\boldsymbol{\omega} = \{\mathbf{W}_i, \mathbf{U}_i, \mathbf{W}_f, \mathbf{U}_f, \mathbf{W}_o, \mathbf{U}_o, \mathbf{W}_g, \mathbf{U}_g\}$ weight matrices and sigm the sigmoid non-linearity. Here an internal state $\mathbf{c}_t$ (also referred to as *cell*) is updated additively.

Alternatively, the model could be re-parametrised as in [Graves et al., 2013]:

$$\begin{pmatrix} \underline{\mathbf{i}} \\ \underline{\mathbf{f}} \\ \underline{\mathbf{o}} \\ \underline{\mathbf{g}} \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \tanh \end{pmatrix} \left(\mathbf{W} \cdot \begin{pmatrix} \mathbf{x}_t \\ \mathbf{h}_{t-1} \end{pmatrix} \right) \tag{3.23}$$

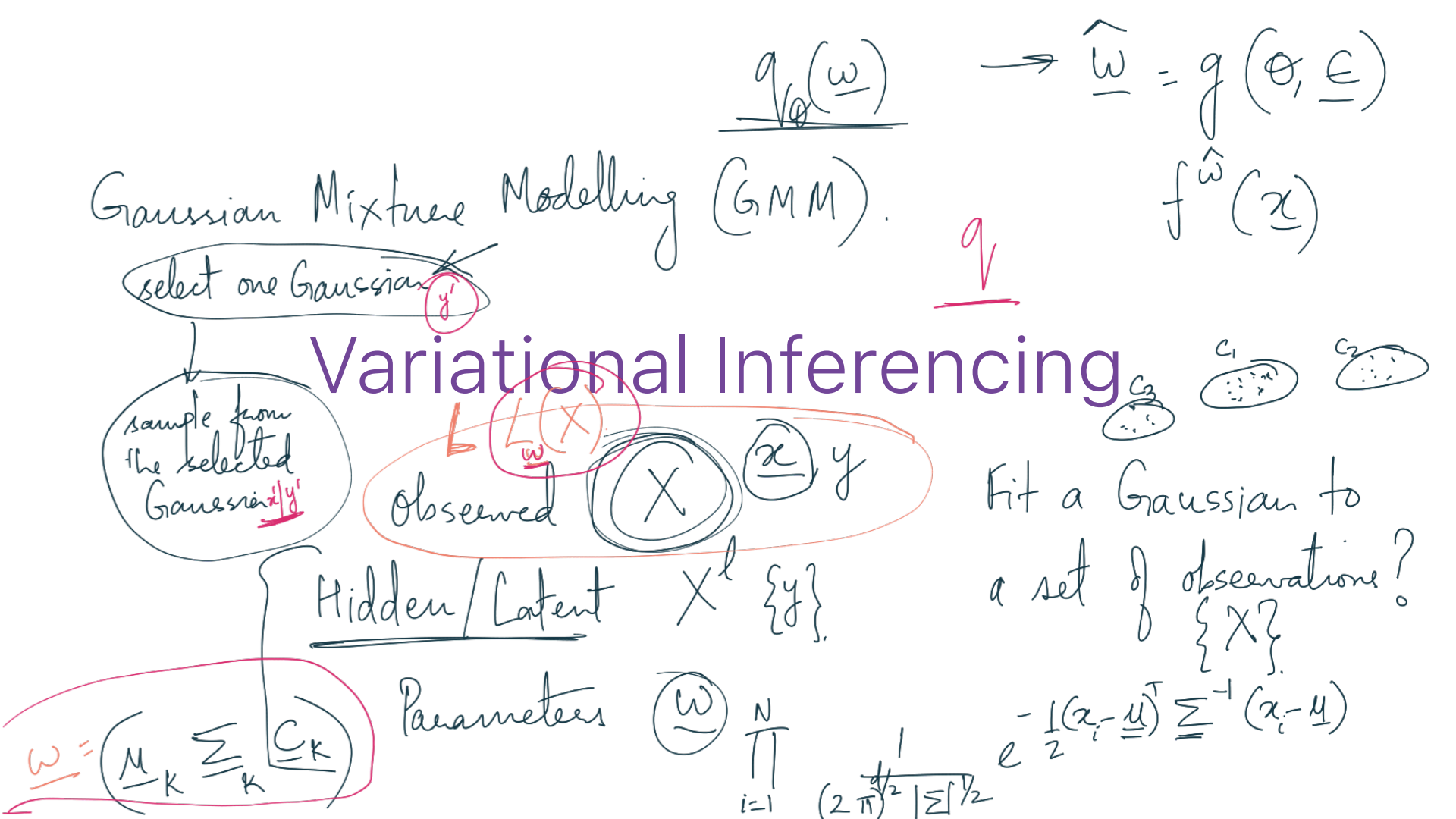
with $\boldsymbol{\omega} = \{\mathbf{W}\}$, $\mathbf{W} = [\mathbf{W}_i, \mathbf{U}_i; \mathbf{W}_f, \mathbf{U}_f; \mathbf{W}_o, \mathbf{U}_o; \mathbf{W}_g, \mathbf{U}_g]$ a matrix of dimensions $4K$ by $2K$ (K being the dimensionality of $\mathbf{x}_t$). We name this parametrisation a *tied-weights* LSTM (compared to the *untied-weights* LSTM parametrisation in eq. (3.22)).

Even though these two parametrisations result in the same deterministic model output, they lead to different approximating distributions $q(\boldsymbol{\omega})$. With the first parametrisation one could use different dropout masks for different gates (even when the same input $\mathbf{x}_t$ is used). This is because the approximating distribution is placed over the matrices rather than the inputs: we might drop certain rows in one weight matrix $\mathbf{W}$ applied to $\mathbf{x}_t$ and different rows in another matrix $\mathbf{W}'$ applied to $\mathbf{x}_t$. With the second parametrisations we would place a distribution over the single matrix $\mathbf{W}$. This leads to a faster forward-pass, but with slightly diminished results (this tradeoff is examined in section §4.5).

Remark (Word embeddings dropout)**.** In datasets with continuous inputs we often apply SRTs such as dropout to the input layer—i.e. to the input vector itself. This is equivalent to placing a distribution over the weight matrix which follows the input and approximately integrating over it (the matrix is optimised, therefore prone to overfitting otherwise).
But for models with discrete inputs such as words (where every word is mapped to a continuous vector—a *word embedding*)—this is seldom done. With word embeddings the input can be seen as either the word embedding itself, or, more conveniently, as a "one-hot" encoding (a vector of zeros with 1 at a single position). The product of the one-hot encoded vector with an embedding matrix $\mathbf{W}_E \in \mathbb{R}^{V \times D}$ (where D is the embedding dimensionality and V is the number of words in the vocabulary) then gives a word embedding. Curiously, this parameter layer is the largest layer in

most language applications, yet it is often not regularised. Since the embedding matrix is optimised it can lead to overfitting, and it is therefore desirable to apply dropout to the one-hot encoded vectors. This in effect is identical to *dropping words at random* throughout the input sentence, and can also be interpreted as encouraging the model to not "depend" on single words for its output.
Note that as before, we randomly set rows of the matrix $\mathbf{W}_E \in \mathbb{R}^{V \times D}$ to zero. Since we repeat the same mask at each time step, we drop the same words throughout the sequence—i.e. we drop word types at random rather than word tokens (as an example, the sentence "the dog and the cat" might become "— dog and — cat" or "the — and the cat", but never "— dog and the cat"). A possible inefficiency implementing this is the requirement to sample V Bernoulli random variables, where V might be large. This can be solved by the observation that for sequences of length T, at most T embeddings could be dropped (other dropped embeddings have no effect on the model output). For $T \ll V$ it is therefore more efficient to first map the words to the word embeddings, and only then to zero-out word embeddings based on their word type.

In the next chapter we will study the techniques above empirically and analyse them quantitatively. This is followed by a survey of recent literature making use of the techniques in real-world problems concerning AI safety, image processing, sequence processing, active learning, and other examples.


$q_\phi(\underline{w})$
$\hat{\underline{w}} = g(\phi, \underline{\epsilon})$
$f^{\hat{w}}(\underline{x})$
Gaussian Mixture Modelling (GMM).
q
Variational Inferencing
select one Gaussian
y'
sample from the selected Gaussian
$x|y'$
$L(X)$
w
Observed
X
$\underline{x}$
y
Hidden / Latent
X^l $\{y\}$.
Parameters
$\underline{w}$
$\underline{w} = (\underline{\mu}_K \; \underline{\Sigma}_K \; \underline{C}_K)$
$\prod_{i=1}^{N} \frac{1}{(2\pi)^{d/2} |\Sigma|^{1/2}} e^{-\frac{1}{2}(x_i - \underline{\mu})^T \underline{\Sigma}^{-1} (x_i - \underline{\mu})}$
Fit a Gaussian to a set of observations?
$\{X\}$.
C_1
C_2
C_3

functional

obs. latent

complete data likelihood.

$$F(\mathbf{q}) = \int \mathbf{q}(X^l) \ln p(X, X^l)\, dX^l + H$$

G

entropy of $q(X^l)$: approximating the posterior over the latent variables.

$$H = -\int \mathbf{q}(X^l) \ln \mathbf{q}(X^l)\, dX^l$$

$$F(\mathbf{q}) = \int \mathbf{q}(X^l) \ln p(X, X^l)\, dX^l - \int \mathbf{q}(X^l) \ln \mathbf{q}(X^l)\, dX^l$$

H

$$F(\mathbf{q}) = \int \mathbf{q}(X^l) \left(\ln p(X, X^l) - \ln \mathbf{q}(X^l) \right) dX^l$$

$$F(\mathbf{q}) = \int \mathbf{q}(X^l) \left(\ln p(X, X^l) - \ln \mathbf{q}(X^l) \right) dX^l$$

$$F(\mathbf{q}) = \int \mathbf{q}(X^l) \left(\ln \frac{p(X, X^l)}{\mathbf{q}(X^l)} \right) dX^l$$

Bayes Rule.

$$F(\mathbf{q}) = \int \mathbf{q}(X^l) \left(\ln \frac{p(X^l \mid X)\, p(X)}{\mathbf{q}(X^l)} \right) dX^l$$

$$F(\mathbf{q}) = \int \mathbf{q}(X^l) \left(\ln \frac{p(X^l \mid X)}{\mathbf{q}(X^l)} \right) dX^l + \int \mathbf{q}(X^l) \left(\ln p(X) \right) dX^l$$

- $F(\mathbf{q}) = \int \mathbf{q}(X^l) \left(\ln \frac{p(X^l \mid X)}{\mathbf{q}(X^l)} \right) dX^l + \int \mathbf{q}(X^l) \left(\ln p(X) \right) dX^l$
- $F(\mathbf{q}) = \int \mathbf{q}(X^l) \left(\ln \frac{p(X^l \mid X)}{\mathbf{q}(X^l)} \right) dX^l + \ln p(X)$
- $F(\mathbf{q}) = -\mathrm{KL}(\mathbf{q} \| p) + \ln p(X)$
- $\ln p(X) = F(\mathbf{q}) + \mathrm{KL}(\mathbf{q} \| p)$

- $\ln p(X) \geq F(\mathbf{q})$

Evidence $\ln p(X; \boldsymbol{\omega}) \geq F(\mathbf{q}; \boldsymbol{\omega})$

Evidence Lower Bound (ELBO).

KL Divergence

- Measures the difference between two probability distributions over the same variable x.
- Let $p(x)$ and $q(x)$ are two probability distributions of a discrete random variable x.
- Both $p(x)$ and $q(x)$ sum to 1, and $p(x) > 0$ and $q(x) > 0$ for any x in X.
- The KL divergence is defined as

$$\mathrm{KL}\left(p(x)\|q(x)\right) = \sum_x p(x)\ \ln\frac{p(x)}{q(x)} = -\sum_x p(x)\ \ln\frac{q(x)}{p(x)}$$

Expectation Maximization

q ω

Example Application (The EM Algorithm)

$F(q, \omega)$

(y) Pick a Gaussian

$P(x|y)$ Sample from the Gaussian

The objective function

(GMM)

- The likelihood of observation $X = x$ given the class label y (Gaussian label y, latent)
- $$P[X = \mathbf{x} \mid Y = y] = \frac{1}{(2\pi)^{d/2} \mid \Sigma_y \mid^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x} - \mu_y)^\top \Sigma_y^{-1} (\mathbf{x} - \mu_y)\right)$$
- The density of X

$$P[X = \mathbf{x}] = \sum_{y=1}^{k} P[Y = \mathbf{y}] P[X = \mathbf{x} \mid Y = y]$$

summing out the joint distribution $P(X, X')$: $\sum_{y=1}^{k} P(X = x, Y = y)$

$$= \sum_{y=1}^{k} c_y \frac{1}{(2\pi)^{d/2} \mid \Sigma_y \mid^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x} - \mu_y)^\top \Sigma_y^{-1} (\mathbf{x} - \mu_y)\right)$$

- $$\log\left(P_{\boldsymbol{\omega}}[X = \mathbf{x}]\right) = \log\left(\sum_{y=1}^{k} P_{\boldsymbol{\omega}}[X = \mathbf{x}, Y = y]\right)$$
- Given an i.i.d training set $S = (\mathbf{x}_1, \ldots, \mathbf{x}_m)$, find parameters $\boldsymbol{\omega}$ that maximise the log-likelihood of S.
- $$L(\boldsymbol{\omega}) = \log \prod_{i=1}^{m} P_{\boldsymbol{\omega}}[X = \mathbf{x}_i] = \sum_{i=1}^{m} \log P_{\boldsymbol{\omega}}[X = \mathbf{x}_i]$$

Likelihood of complete data.

$$= \sum_{i=1}^{m} \log\left(\sum_{y=1}^{k} P_{\boldsymbol{\omega}}[X = \mathbf{x}_i, Y = y]\right)$$

intractable · log acting on summation.

- The maximum likelihood estimator is the solution of the maximisation problem

- $$\arg\max_{\boldsymbol{\omega}} L(\boldsymbol{\omega}) = \arg\max_{\boldsymbol{\omega}} \sum_{i=1}^{m} \log \left(\sum_{y=1}^{k} P_{\boldsymbol{\omega}}[X = \mathbf{x}_i, Y = y] \right)$$

k: # Gaussians in the GM

- We formulate the expectation of the complete data log likelihood with respect to the posterior distribution over the latent/hidden variables.

$$G(\mathbf{q}, \boldsymbol{\omega}) = \sum_{i=1}^{m} \sum_{y=1}^{k} \mathbf{q}_{i,y} \log \left(P_{\boldsymbol{\omega}}[X = \mathbf{x}_i, Y = y] \right)$$

$\omega = \{c_y, \mu_y, \Sigma_y\}_{\forall y}$

$q_{iy} = P[Y = y \mid X = x_i]$

	y_1	y_2	y_3 .. y_k
x_1	—	—	
x_2			
$\vdots$			
x_i			$p(y_k \mid x_i)$
$\vdots$			
x_m			

- The function is formulated with the assumption that the optimisation $\arg\max_{\boldsymbol{\omega}} G(\mathbf{q}, \boldsymbol{\omega})$ is tractable
- The objective function to be optimised w.r.t. the latent variables is

$$F(\mathbf{q}, \boldsymbol{\omega}) = G(\mathbf{q}, \boldsymbol{\omega}) - \sum_{i=1}^{m} \sum_{y=1}^{k} \mathbf{q}_{i,y} \log(\mathbf{q}_{i,y})$$

alternative to $L_w(X)$

$$F(\mathbf{q}, \boldsymbol{\omega}) = G(\mathbf{q}, \boldsymbol{\omega}) + H(\mathbf{q})$$

$$F(\mathbf{q}, \boldsymbol{\omega}) = \sum_{i=1}^{m} \sum_{y=1}^{k} \mathbf{q}_{i,y} \log\left(P_{\boldsymbol{\omega}}[X = \mathbf{x}_i, Y = y]\right) - \sum_{i=1}^{m} \sum_{y=1}^{k} \mathbf{q}_{i,y} \log(\mathbf{q}_{i,y})$$

ELBO

$$
\begin{aligned}
F(\mathbf{q}, \boldsymbol{\omega}) &= \sum_{i=1}^{m} \sum_{y=1}^{k} \mathbf{q}_{i,y} \log \left(P_{\boldsymbol{\omega}}[X = \mathbf{x}_i, Y = y] \right) - \sum_{i=1}^{m} \sum_{y=1}^{k} \mathbf{q}_{i,y} \log(\mathbf{q}_{i,y}) \\
&= \sum_{i=1}^{m} \sum_{y=1}^{k} \mathbf{q}_{i,y} \log \left(\frac{P_{\boldsymbol{\omega}}[X = \mathbf{x}_i, Y = y]}{\mathbf{q}_{i,y}} \right) \\
&\leq \sum_{i=1}^{m} \left(\log \left(\sum_{y=1}^{k} \mathbf{q}_{i,y} \frac{P_{\boldsymbol{\omega}}[X = \mathbf{x}_i, Y = y]}{\mathbf{q}_{i,y}} \right) \right) \\
&= \sum_{i=1}^{m} \left(\log \left(P_{\boldsymbol{\omega}}[X = \mathbf{x}_i] \right) \right) = L(\boldsymbol{\omega})
\end{aligned}
$$

ELBO

A

B

$\log A - \log B$

$\log \frac{A}{B}$

$\sum_y x_y f(z_y) \leq f\left(\sum_y x_y z_y\right)$

observed data log likelihood.

1) Expectation Step: (q) 2) Maximization Step (ω)

- For a given $\boldsymbol{\omega}$, function $F(\mathbf{q}, \boldsymbol{\omega})$ attains its maximum value for

$$\mathbf{q}_{i,y} = P_{\boldsymbol{\omega}}[Y = y \,|\, X = \mathbf{x}_i]$$ true posterior

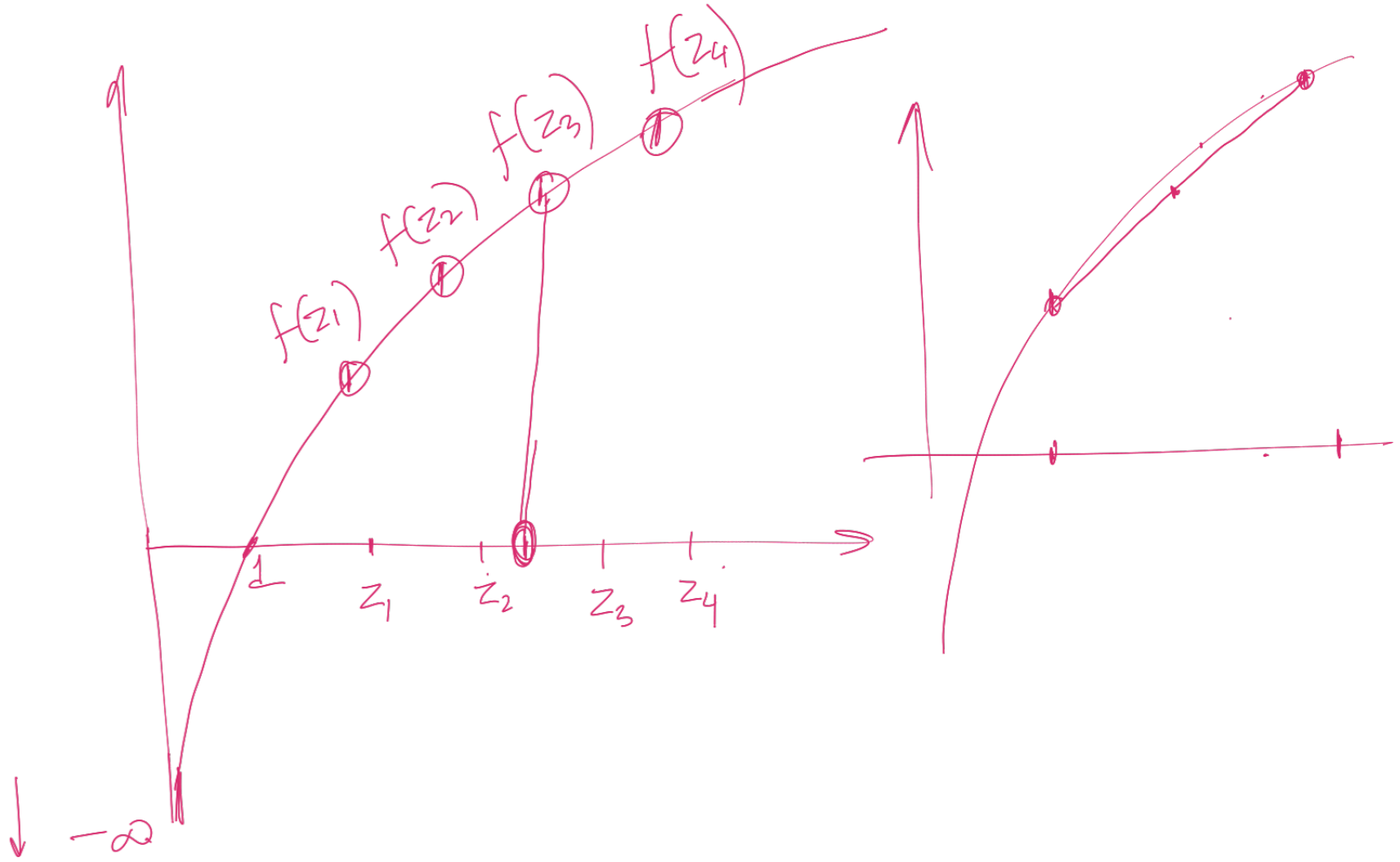
- $$F(\mathbf{q}, \boldsymbol{\omega}) = \sum_{i=1}^{m} \left(\sum_{y=1}^{k} P_{\boldsymbol{\omega}}[Y = y \,|\, X = \mathbf{x}_i] \; \log \left(\frac{P_{\boldsymbol{\omega}}[Y = y, X = \mathbf{x}_i]}{P_{\boldsymbol{\omega}}[Y = y \,|\, X = \mathbf{x}_i]} \right) \right)$$

$$= \sum_{i=1}^{m} \sum_{y=1}^{k} P_{\boldsymbol{\omega}}[Y = y \,|\, X = \mathbf{x}_i] \; \log \left(P_{\boldsymbol{\omega}}[X = \mathbf{x}_i] \right)$$

$$= \sum_{i=1}^{m} \log \left(P_{\boldsymbol{\omega}}[X = \mathbf{x}_i] \right) \sum_{y=1}^{k} P_{\boldsymbol{\omega}}[Y = y \,|\, X = \mathbf{x}_i]$$

$$= \sum_{i=1}^{m} \log \left(P_{\boldsymbol{\omega}}[X = \mathbf{x}_i] \right) = L(\boldsymbol{\omega})$$

log likelihood of observed data


f(z4)
f(z3)
f(z2)
f(z1)
1
z1
z2
z3
z4
−∞

arXiv:1704.02685v2 [cs.CV] 12 Oct 2019

Learning Important Features Through Propagating Activation Differences

Avanti Shrikumar [1] **Peyton Greenside** [1] **Anshul Kundaje** [1]

Abstract

The purported "black box" nature of neural networks is a barrier to adoption in applications where interpretability is essential. Here we present DeepLIFT (Deep Learning Important FeaTures), a method for decomposing the output prediction of a neural network on a specific input by backpropagating the contributions of all neurons in the network to every feature of the input. DeepLIFT compares the activation of each neuron to its 'reference activation' and assigns contribution scores according to the difference. By optionally giving separate consideration to positive and negative contributions, DeepLIFT can also reveal dependencies which are missed by other approaches. Scores can be computed efficiently in a single backward pass. We apply DeepLIFT to models trained on MNIST and simulated genomic data, and show significant advantages over gradient-based methods. Video tutorial: http://goo.gl/qKb7pL, ICML slides: bit.ly/deeplifticmlslides, ICML talk: https://vimeo.com/238275076, code: http://goo.gl/RM8jvH.

1. Introduction

As neural networks become increasingly popular, their black box reputation is a barrier to adoption when interpretability is paramount. Here, we present DeepLIFT (Deep Learning Important FeaTures), a novel algorithm to assign importance score to the inputs for a given output. Our approach is unique in two regards. First, it frames the question of importance in terms of differences from a 'reference' state, where the 'reference' is chosen according to the problem at hand. In contrast to most gradient-based methods, using a difference-from-reference allows DeepLIFT to propagate an importance signal even in situations where the gradient is zero and avoids artifacts caused by discontinuities in the gradient. Second, by optionally giving separate consideration to the effects of positive and negative contributions at nonlinearities, DeepLIFT can reveal dependencies missed by other approaches. As DeepLIFT scores are computed using a backpropagation-like algorithm, they can be obtained efficiently in a single backward pass after a prediction has been made.

[1]Stanford University, Stanford, California, USA. Correspondence: A Kundaje <akundaje@stanford.edu>.

2. Previous Work

This section provides a review of existing approaches to assign importance scores for a given task and input example.

2.1. Perturbation-Based Forward Propagation Approaches

These approaches make perturbations to individual inputs or neurons and observe the impact on later neurons in the network. Zeiler & Fergus (Zeiler & Fergus, 2013) occluded different segments of an input image and visualized the change in the activations of later layers. "In-silico mutagenesis" (Zhou & Troyanskaya, 2015) introduced virtual mutations at individual positions in a genomic sequence and quantified the their impact on the output. Zintgraf et al. (Zintgraf et al., 2017) proposed a clever strategy for analyzing the difference in a prediction after marginalizing over each input patch. However, such methods can be computationally inefficient as each perturbation requires a separate forward propagation through the network. They may also underestimate the importance of features that have saturated their contribution to the output (**Fig. 1**).

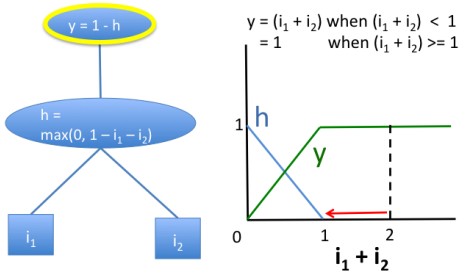


Figure 1. **Perturbation-based approaches and gradient-based approaches fail to model saturation**. Illustrated is a simple network exhibiting saturation in the signal from its inputs. At the point where $i_1 = 1$ and $i_2 = 1$, perturbing either i_1 or i_2 to 0 will not produce a change in the output. Note that the gradient of the output w.r.t the inputs is also zero when $i_1 + i_2 > 1$.

2.2. Backpropagation-Based Approaches

Unlike perturbation methods, backpropagation approaches propagate an importance signal from an output neuron backwards through the layers to the input in one pass, making them efficient. DeepLIFT is one such approach.

2.2.1. Gradients, Deconvolutional Networks and Guided Backpropagation

Simonyan et al. (Simonyan et al., 2013) proposed using the gradient of the output w.r.t. pixels of an input image to compute a "saliency map" of the image in the context of image classification tasks. The authors showed that this was similar to deconvolutional networks (Zeiler & Fergus, 2013) except for the handling of the nonlinearity at rectified linear units (ReLUs). When backpropagating importance using gradients, the gradient coming into a ReLU during the backward pass is zero'd out if the input to the ReLU during the forward pass is negative. By contrast, when backpropagating an importance signal in deconvolutional networks, the importance signal coming into a ReLU during the backward pass is zero'd out if and only if it is negative, with no regard to sign of the input to the ReLU during the forward pass. Springenberg et al., (Springenberg et al., 2014) combined these two approaches into Guided Backpropagation, which zero's out the importance signal at a ReLU if either the input to the ReLU during the forward pass is negative or the importance signal during the backward pass is negative. Guided Backpropagation can be thought of as equivalent to computing gradients, with the caveat that any gradients that become negative during the backward pass are discarded at ReLUs. Due to the zero-ing out of negative gradients, both guided backpropagation and deconvolutional networks can fail to highlight inputs that contribute negatively to the output. Additionally, none of the three approaches would address the saturation problem illustrated in **Fig. 1**, as the gradient of y w.r.t. h is negative (causing Guided Backprop and deconvolutional networks to assign zero importance), and the gradient of h w.r.t both i_1 and i_2 is zero when $i_1 + i_2 > 1$ (causing both gradients and Guided Backprop to be zero). Discontinuities in the gradients can also cause undesirable artifacts (**Fig. 2**).

2.2.2. Layerwise Relevance Propagation and Gradient $\times$ Input

Bach et al. (Bach et al., 2015) proposed an approach for propagating importance scores called Layerwise Relevance Propagation (LRP). Shrikumar et al. and Kindermans et al. (Shrikumar et al., 2016; Kindermans et al., 2016) showed that absent modifications to deal with numerical stability, the LRP rules for ReLU networks were equivalent within a scaling factor to an elementwise product between the saliency maps of Simonyan et al. and the input (in other words, gradient $\times$ input). In our experiments, we compare DeepLIFT to gradient $\times$ input as the latter is easily implemented on a GPU, whereas (at the time of writing) LRP did not have a GPU implementation available to our knowledge.

While gradient $\times$ input is often preferable to gradients alone as it leverages the sign and strength of the input, it still does not address the saturation problem in **Fig. 1** or the thresholding artifact in **Fig. 2**.

2.2.3. Integrated Gradients

Instead of computing the gradients at only the current value of the input, one can integrate the gradients as the inputs are scaled up from some starting value (eg: all zeros) to their current value (Sundararajan et al., 2016). This addressess the saturation and thresholding problems of **Fig. 1** and **Fig. 2**, but numerically obtaining high-quality integrals adds computational overhead. Further, this approach can still give misleading results (see **Section 3.4.3**).

2.3. Grad-CAM and Guided CAM

Grad-CAM (Selvaraju et al., 2016) computes a coarse-grained feature-importance map by associating the feature maps in the final convolutional layer with particular classes based on the gradients of each class w.r.t. each feature map, and then using the weighted activations of the feature maps as an indication of which inputs are most important. To obtain more fine-grained feature importance, the authors proposed performing an elementwise product between the scores obtained from Grad-CAM and the scores obtained from Guided Backpropagation, termed Guided Grad-CAM. However, this strategy inherits the limitations of Guided Backpropagation caused by zero-ing out negative gradients during backpropagation. It is also specific to convolutional neural networks.

3. The DeepLIFT Method

3.1. The DeepLIFT Philosophy

DeepLIFT explains the difference in output from some 'reference' output in terms of the difference of the input from some 'reference' input. The 'reference' input represents some default or 'neutral' input that is chosen according to what is appropriate for the problem at hand (see **Section 3.3** for more details). Formally, let t represent some target output neuron of interest and let $x_1, x_2, ..., x_n$ represent some neurons in some intermediate layer or set of layers that are necessary and sufficient to compute t. Let t^0 represent the reference activation of t. We define the quantity Δt to be the difference-from-reference, that is $\Delta t = t - t^0$. DeepLIFT assigns contribution scores $C_{\Delta x_i \Delta t}$ to Δx_i s.t.:

$$\sum_{i=1}^{n} C_{\Delta x_i \Delta t} = \Delta t \tag{1}$$

We call **Eq. 1** the **summation-to-delta** property. $C_{\Delta x_i \Delta t}$ can be thought of as the amount of difference-from-reference in t that is attributed to or 'blamed' on the difference-from-reference of x_i. Note that when a neuron's transfer function is well-behaved, the output is locally linear in its inputs, providing additional motivation for **Eq. 1**.

$C_{\Delta x_i \Delta t}$ can be non-zero even when $\frac{\partial t}{\partial x_i}$ is zero. This allows DeepLIFT to address a fundamental limitation of gradients because, as illustrated in **Fig. 1**, a neuron can be signaling meaningful information even in the regime where its gradient is zero. Another drawback of gradients addressed by DeepLIFT is illustrated in **Fig. 2**, where the discontinuous nature of gradients causes sudden jumps in the importance score over infinitesimal changes in the input. By contrast, the difference-from-reference is continuous, allowing DeepLIFT to avoid discontinuities caused by bias terms.

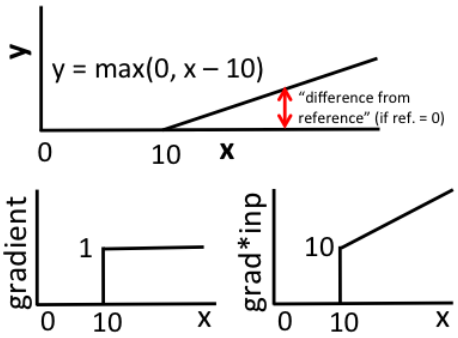


Figure 2. **Discontinuous gradients can produce misleading importance scores**. Response of a single rectified linear unit with a bias of -10. Both gradient and gradient$\times$input have a discontinuity at $x = 10$; at $x = 10 + \epsilon$, gradient$\times$input assigns a contribution of $10 + \epsilon$ to x and -10 to the bias term (ϵ is a small positive number). When $x < 10$, contributions on x and the bias term are both 0. By contrast, the difference-from-reference (red arrow, top figure) gives a continuous increase in the contribution score.

3.2. Multipliers and the Chain Rule

3.2.1. Definition of Multipliers

For a given input neuron x with difference-from-reference Δx, and target neuron t with difference-from-reference Δt that we wish to compute the contribution to, we define the multiplier $m_{\Delta x \Delta t}$ as:

$$m_{\Delta x \Delta t} = \frac{C_{\Delta x \Delta t}}{\Delta x} \tag{2}$$

In other words, the multiplier $m_{\Delta x \Delta t}$ is the contribution of Δx to Δt divided by Δx. Note the close analogy to the idea of partial derivatives: the partial derivative $\frac{\partial t}{\partial x}$ is the infinitesimal change in t caused by an infinitesimal change in x, divided by the infinitesimal change in x. The multiplier is similar in spirit to a partial derivative, but over finite differences instead of infinitesimal ones.

3.2.2. The Chain Rule For Multipliers

Assume we have an input layer with neurons $x_1, ..., x_n$, a hidden layer with neurons $y_1, ..., y_n$, and some target output neuron t. Given values for $m_{\Delta x_i \Delta y_j}$ and $m_{\Delta y_j \Delta t}$, the following definition of $m_{\Delta x_i \Delta t}$ is consistent with the summation-to-delta property in **Eq. 1** (see **Appendix A** for the proof):

$$m_{\Delta x_i \Delta t} = \sum_j m_{\Delta x_i \Delta y_j} m_{\Delta y_j \Delta t} \tag{3}$$

We refer to **Eq. 3** as the **chain rule for multipliers**. Given the multipliers for each neuron to its immediate successors, we can compute the multipliers for any neuron to a given target neuron efficiently via backpropagation - analogous to how the chain rule for partial derivatives allows us to compute the gradient w.r.t. the output via backpropagation.

3.3. Defining the Reference

When formulating the DeepLIFT rules described in **Section 3.5**, we assume that the reference of a neuron is its activation on the reference input. Formally, say we have a neuron y with inputs $x_1, x_2, ...$ such that $y = f(x_1, x_2, ...)$. Given the reference activations $x_1^0, x_2^0, ...$ of the inputs, we can calculate the reference activation y^0 of the output as:

$$y^0 = f(x_1^0, x_2^0, ...) \tag{4}$$

i.e. references for all neurons can be found by choosing a reference input and propagating activations through the net.

The choice of a reference input is critical for obtaining insightful results from DeepLIFT. In practice, choosing a good reference would rely on domain-specific knowledge, and in some cases it may be best to compute DeepLIFT scores against multiple different references. As a guiding principle, we can ask ourselves "what am I interested in measuring differences against?". For MNIST, we use a reference input of all-zeros as this is the background of the images. For the binary classification tasks on DNA sequence inputs (strings over the alphabet {A,C,G,T}), we obtained sensible results using either a reference input containing the expected frequencies of ACGT in the background (**Fig. 5**), or by averaging the results over multiple reference inputs for each sequence that are generated by shuffling each original sequence (**Appendix J**). For CIFAR10 data, we found that using a blurred version of the original image as the

reference highlighted outlines of key objects, while an all-zeros reference highlighted hard-to-interpret pixels in the background (**Appendix L**).

It is important to note that gradient×input implicitly uses a reference of all-zeros (it is equivalent to a first-order Taylor approximation of gradient×Δinput where Δ is measured w.r.t. an input of zeros). Similary, integrated gradients (**Section 2.2.3**) requires the user to specify a starting point for the integral, which is conceptually similar to specifying a reference for DeepLIFT. While Guided Backprop and pure gradients don't use a reference, we argue that this is a limitation as these methods only describe the local behaviour of the output at the specific input value, without considering how the output behaves over a range of inputs.

3.4. Separating Positive and Negative Contributions

We will see in **Section 3.5.3** that, in some situations, it is essential to treat positive and negative contributions differently. To do this, for every neuron y, we will introduce Δy^+ and Δy^- to represent the positive and negative components of Δy, such that:

$$\Delta y = \Delta y^+ + \Delta y^-$$
$$C_{\Delta y \Delta t} = C_{\Delta y^+ \Delta t} + C_{\Delta y^- \Delta t}$$

For linear neurons, Δy^+ and Δy^- are found by writing Δy as a sum of terms involving its inputs Δx_i and grouping positive and negative terms together. The importance of this will become apparent when applying the RevealCancel rule (**Section 3.5.3**), where for a given target neuron t we may find that $m_{\Delta y^+ \Delta t}$ and $m_{\Delta y^- \Delta t}$ differ. However, when applying only the Linear or Rescale rules (**Section 3.5.1** and **Section 3.5.2**), $m_{\Delta y \Delta t} = m_{\Delta y^+ \Delta t} = m_{\Delta y^- \Delta t}$.

3.5. Rules for Assigning Contribution Scores

We present the rules for assigning contribution scores for each neuron to its immediate inputs. In conjunction with the chain rule for multipliers (**Section 3.2**), these rules can be used to find the contributions of any input (not just the immediate inputs) to a target output via backpropagation.

3.5.1. The Linear Rule

This applies to Dense and Convolutional layers (excluding nonlinearities). Let y be a linear function of its inputs x_i such that $y = b + \sum_i w_i x_i$. We have $\Delta y = \sum_i w_i \Delta x_i$. We define the positive and negative parts of Δy as:

$$\Delta y^+ = \sum_i 1\{w_i \Delta x_i > 0\} w_i \Delta x_i$$
$$= \sum_i 1\{w_i \Delta x_i > 0\} w_i (\Delta x_i^+ + \Delta x_i^-)$$

$$\Delta y^- = \sum_i 1\{w_i \Delta x_i < 0\} w_i \Delta x_i$$
$$= \sum_i 1\{w_i \Delta x_i < 0\} w_i (\Delta x_i^+ + \Delta x_i^-)$$

Which leads to the following choice for the contributions:

$$C_{\Delta x_i^+ \Delta y^+} = 1\{w_i \Delta x_i > 0\} w_i \Delta x_i^+$$
$$C_{\Delta x_i^- \Delta y^+} = 1\{w_i \Delta x_i > 0\} w_i \Delta x_i^-$$
$$C_{\Delta x_i^+ \Delta y^-} = 1\{w_i \Delta x_i < 0\} w_i \Delta x_i^+$$
$$C_{\Delta x_i^- \Delta y^-} = 1\{w_i \Delta x_i < 0\} w_i \Delta x_i^-$$

We can then find multipliers using the definition in **Section 3.2.1**, which gives $m_{\Delta x_i^+ \Delta y^+} = m_{\Delta x_i^- \Delta y^+} = 1\{w_i \Delta x_i > 0\} w_i$ and $m_{\Delta x_i^+ \Delta y^-} = m_{\Delta x_i^- \Delta y^-} = 1\{w_i \Delta x_i < 0\} w_i$.

What about when $\Delta x_i = 0$? While setting multipliers to 0 in this case would be consistent with summation-to-delta, it is possible that Δx_i^+ and Δx_i^- are nonzero (and cancel each other out), in which case setting the multiplier to 0 would fail to propagate importance to them. To avoid this, we set $m_{\Delta x_i^+ \Delta y^+} = m_{\Delta x_i^+ \Delta y^-} = 0.5 w_i$ when Δx_i is 0 (similarly for Δx^-). See **Appendix B** for how to compute these multipliers using standard neural network ops.

3.5.2. The Rescale Rule

This rule applies to nonlinear transformations that take a single input, such as the ReLU, tanh or sigmoid operations. Let neuron y be a nonlinear transformation of its input x such that $y = f(x)$. Because y has only one input, we have by summation-to-delta that $C_{\Delta x \Delta y} = \Delta y$, and consequently $m_{\Delta x \Delta y} = \frac{\Delta y}{\Delta x}$. For the Rescale rule, we set Δy^+ and Δy^- proportional to Δx^+ and Δx^- as follows:

$$\Delta y^+ = \frac{\Delta y}{\Delta x} \Delta x^+ = C_{\Delta x^+ \Delta y^+}$$
$$\Delta y^- = \frac{\Delta y}{\Delta x} \Delta x^- = C_{\Delta x^- \Delta y^-}$$

Based on this, we get:

$$m_{\Delta x^+ \Delta y^+} = m_{\Delta x^- \Delta y^-} = m_{\Delta x \Delta y} = \frac{\Delta y}{\Delta x}$$

In the case where $x \to x^0$, we have $\Delta x \to 0$ and $\Delta y \to 0$. The definition of the multiplier approaches the derivative, i.e. $m_{\Delta x \Delta y} \to \frac{dy}{dx}$, where the $\frac{dy}{dx}$ is evaluated at $x = x^0$. We can thus use the gradient instead of the multiplier when x is close to its reference to avoid numerical instability issues caused by having a small denominator.

Note that the Rescale rule addresses both the saturation and the thresholding problems illustrated in **Fig. 1** and **Fig. 2**. In the case of **Fig. 1**, if $i_1^0 = i_2^0 = 0$, then at $i_1 + i_2 > 1$ we have $\Delta h = -1$ and $\Delta y = 1$, giving

$m_{\Delta h \Delta y} = \frac{\Delta y}{\Delta h} = -1$ even though $\frac{dy}{dh} = 0$ (in other words, using difference-from-reference allows information to flow even when the gradient is zero). In the case of **Fig. 2**, assuming $x^0 = y^0 = 0$, at $x = 10 + \epsilon$ we have $\Delta y = \epsilon$, giving $m_{\Delta x \Delta y} = \frac{\epsilon}{10+\epsilon}$ and $C_{\Delta x \Delta y} = \Delta x \times m_{\Delta x \Delta y} = \epsilon$. By contrast, gradient×input assigns a contribution of $10+\epsilon$ to x and -10 to the bias term (DeepLIFT never assigns importance to bias terms).

As revealed in previous work (Lundberg & Lee, 2016), there is a connection between DeepLIFT and Shapely values. Briefly, the Shapely values measure the average marginal effect of including an input over all possible orderings in which inputs can be included. If we define "including" an input as setting it to its actual value instead of its reference value, DeepLIFT can be thought of as a fast approximation of the Shapely values. At the time, Lundberg & Lee cited a preprint of DeepLIFT which described only the Linear and Rescale rules with no separate treatment of positive and negative contributions.

3.5.3. An Improved Approximation of the Shapely Values: The RevealCancel Rule

While the Rescale rule improves upon simply using gradients, there are still some situations where it can provide misleading results. Consider the $\min(i_1, i_2)$ operation depicted in **Fig. 3**, with reference values of $i_1 = 0$ and $i_2 = 0$. Using the Rescale rule, all importance would be assigned either to i_1 or to i_2 (whichever is smaller). This can obscure the fact that both inputs are relevant for the $\min$ operation.

To understand why this occurs, consider the case when $i_1 > i_2$. We have $h_1 = (i_1 - i_2) > 0$ and $h_2 = \max(0, h_1) = h_1$. By the Linear rule, we calculate that $C_{\Delta i_1 \Delta h_1} = i_1$ and $C_{\Delta i_2 \Delta h_1} = -i_2$. By the Rescale rule, the multiplier $m_{\Delta h_1 \Delta h_2}$ is $\frac{\Delta h_2}{\Delta h_1} = 1$, and thus $C_{\Delta i_1 \Delta h_2} = m_{\Delta h_1 \Delta h_2} C_{\Delta i_1 \Delta h_1} = i_1$ and $C_{\Delta i_2 \Delta h_2} = m_{\Delta h_1 \Delta h_2} C_{\Delta i_2 \Delta h_1} = -i_2$. The total contribution of i_1 to the output o becomes $(i_1 - C_{\Delta i_1 \Delta h_2}) = (i_1 - i_1) = 0$, and the total contribution of i_2 to o is $-C_{\Delta i_2 \Delta h_2} = i_2$. This calculation is misleading as it discounts the fact that $C_{\Delta i_2 \Delta h_2}$ would be 0 if i_1 were 0 - in other words, it ignores a dependency induced between i_1 and i_2 that comes from i_2 canceling out i_1 in the nonlinear neuron h_2. A similar failure occurs when $i_1 < i_2$; the Rescale rule results in $C_{\Delta i_1 \Delta o} = i_1$ and $C_{\Delta i_2 \Delta o} = 0$. Note that gradients, gradient×input, Guided Backpropagation and integrated gradients would also assign all importance to either i_1 or i_2, because for any given input the gradient is zero for one of i_1 or i_2 (see **Appendix C** for a detailed calculation).

One way to address this is by treating the positive and negative contributions separately. We again consider the nonlinear neuron $y = f(x)$. Instead of assuming that Δy^+ and Δy^- are proportional to Δx^+ and Δx^- and that $m_{\Delta x^+ \Delta y^+} = m_{\Delta x^- \Delta y^-} = m_{\Delta x \Delta y}$ (as is done for the Rescale rule), we define them as follows:

$$\Delta y^+ = \frac{1}{2}\left(f(x^0 + \Delta x^+) - f(x^0)\right) + \frac{1}{2}\left(f(x^0 + \Delta x^- + \Delta x^+) - f(x^0 + \Delta x^-)\right)$$

$$\Delta y^- = \frac{1}{2}\left(f(x^0 + \Delta x^-) - f(x^0)\right) + \frac{1}{2}\left(f(x^0 + \Delta x^+ + \Delta x^-) - f(x^0 + \Delta x^+)\right)$$

$$m_{\Delta x^+ \Delta y^+} = \frac{C_{\Delta x^+ y^+}}{\Delta x^+} = \frac{\Delta y^+}{\Delta x^+}; m_{\Delta x^- \Delta y^-} = \frac{\Delta y^-}{\Delta x^-}$$

In other words, we set Δy^+ to the average impact of Δx^+ after no terms have been added and after Δx^- has been added, and we set Δy^- to the average impact of Δx^- after no terms have been added and after Δx^+ has been added. This can be thought of as the Shapely values of Δx^+ and Δx^- contributing to y.

By considering the impact of the positive terms in the absence of negative terms, and the impact of negative terms in the absence of positive terms, we alleviate some of the issues that arise from positive and negative terms canceling each other out. In the case of **Fig. 3**, RevealCancel would assign a contribution of $0.5 \min(i_1, i_2)$ to both inputs (see **Appendix C** for a detailed calculation).

While the RevealCancel rule also avoids the saturation and thresholding pitfalls illustrated in **Fig. 1** and **Fig. 2**, there are some circumstances where we might prefer to use the Rescale rule. Specifically, consider a thresholded ReLU where $\Delta y > 0$ iff $\Delta x \geq b$. If $\Delta x < b$ merely indicates noise, we would want to assign contributions of 0 to both Δx^+ and Δx^- (as done by the Rescale rule) to mitigate the noise. RevealCancel may assign nonzero contributions by considering Δx^+ in the absence of Δx^- and vice versa.

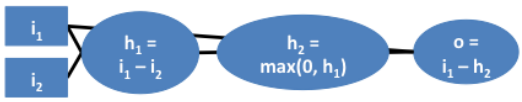


Figure 3. Network computing $o = \min(i_1, i_2)$. Assume $i_1^0 = i_2^0 = 0$. When $i_1 < i_2$ then $\frac{dy}{di_2} = 0$, and when $i_2 < i_1$ then $\frac{do}{di_1} = 0$. Using any of the backpropagation approaches described in **Section 2.2** would result in importance assigned either exclusively to i_1 or i_2. With the RevealCancel rule, the net assigns $0.5 \min(i_1, i_2)$ importance to both inputs.

3.6. Choice of Target Layer

In the case of softmax or sigmoid outputs, we may prefer to compute contributions to the linear layer preceding the final nonlinearity rather than the final nonlinearity itself. This would be to avoid an attentuation caused by the

summation-to-delta property described in **Section 3.1**. For example, consider a sigmoid output $o = \sigma(y)$, where y is the logit of the sigmoid function. Assume $y = x_1 + x_2$, where $x_1^0 = x_2^0 = 0$. When $x_1 = 50$ and $x_2 = 0$, the output o saturates at very close to 1 and the contributions of x_1 and x_2 are 0.5 and 0 respectively. However, when $x_1 = 100$ and $x_2 = 100$, the output o is still very close to 1, but the contributions of x_1 and x_2 are now both 0.25. This can be misleading when comparing scores across different inputs because a stronger contribution to the logit would not always translate into a higher DeepLIFT score. To avoid this, we compute contributions to y rather than o.

Adjustments for Softmax Layers

If we compute contributions to the linear layer preceding the softmax rather than the softmax output, an issue that could arise is that the final softmax output involves a normalization over all classes, but the linear layer before the softmax does not. To address this, we can normalize the contributions to the linear layer by subtracting the mean contribution to all classes. Formally, if n is the number of classes, $C_{\Delta x \Delta c_i}$ represents the unnormalized contribution to class c_i in the linear layer and $C'_{\Delta x \Delta c_i}$ represents the normalized contribution, we have:

$$C'_{\Delta x \Delta c_i} = C_{\Delta x \Delta c_i} - \frac{1}{n} \sum_{j=1}^{n} C_{\Delta x \Delta c_j} \tag{5}$$

As a justification for this normalization, we note that subtracting a fixed value from all the inputs to the softmax leaves the output of the softmax unchanged.

4. Results

4.1. Digit Classification (MNIST)

We train a convolutional neural network on MNIST (LeCun et al., 1999) using Keras (Chollet, 2015) to perform digit classification and obtain 99.2% test-set accuracy. The architecture consists of two convolutional layers, followed by a fully connected layer, followed by the softmax output layer (see **Appendix D** for full details on model architecture and training). We used convolutions with stride > 1 instead of pooling layers, which did not result in a drop in performance as is consistent with previous work (Springenberg et al., 2014). For DeepLIFT and integrated gradients, we used a reference input of all zeros.

To evaluate importance scores obtained by different methods, we design the following task: given an image that originally belongs to class c_o, we identify which pixels to erase to convert the image to some target class c_t. We do this by finding $S_{x_i \text{diff}} = S_{x_i c_o} - S_{x_i c_t}$ (where $S_{x_i c}$ is the score for pixel x_i and class c) and erasing up to 157 pixels (20% of the image) ranked in descending order of $S_{x_i \text{diff}}$ for which $S_{x_i \text{diff}} > 0$. We then evaluate the change in the log-odds score between classes c_o and c_t for the original image and the image with the pixels erased.

As shown in **Fig. 4**, DeepLIFT with the RevealCancel rule outperformed the other backpropagation-based methods. Integrated gradients (**Section 2.2.3**) computed numerically over either 5 or 10 intervals produced results comparable to each other, suggesting that adding more intervals would not change the result. Integrated gradients also performed comparably to gradient*input, suggesting that saturation and thresholding failure modes are not common on MNIST data. Guided Backprop discards negative gradients during backpropagation, perhaps explaining its poor performance at discriminating between classes. We also explored using the Rescale rule instead of RevealCancel on various layers and found that it degraded performance (**Appendix E**).

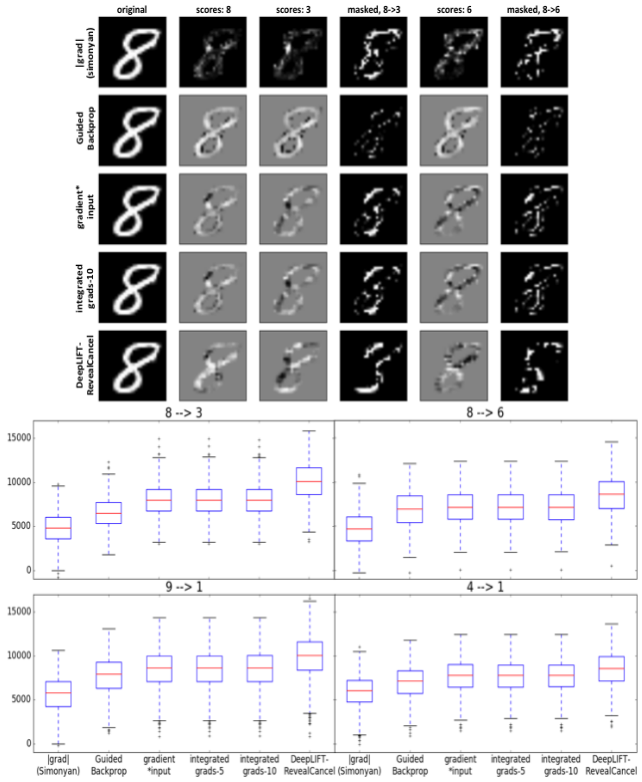


Figure 4. **DeepLIFT with the RevealCancel rule better identifies pixels to convert one digit to another.** Top: result of masking pixels ranked as most important for the original class (8) relative to the target class (3 or 6). Importance scores for class 8, 3 and 6 are also shown. The selected image had the highest change in log-odds scores for the 8→6 conversion using gradient*input or integrated gradients to rank pixels. Bottom: boxplots of increase in log-odds scores of target vs. original class after the mask is applied, for 1K images belonging to the original class in the testing set. "Integrated gradients-n" refers to numerically integrating the gradients over n evenly-spaced intervals using the midpoint rule.

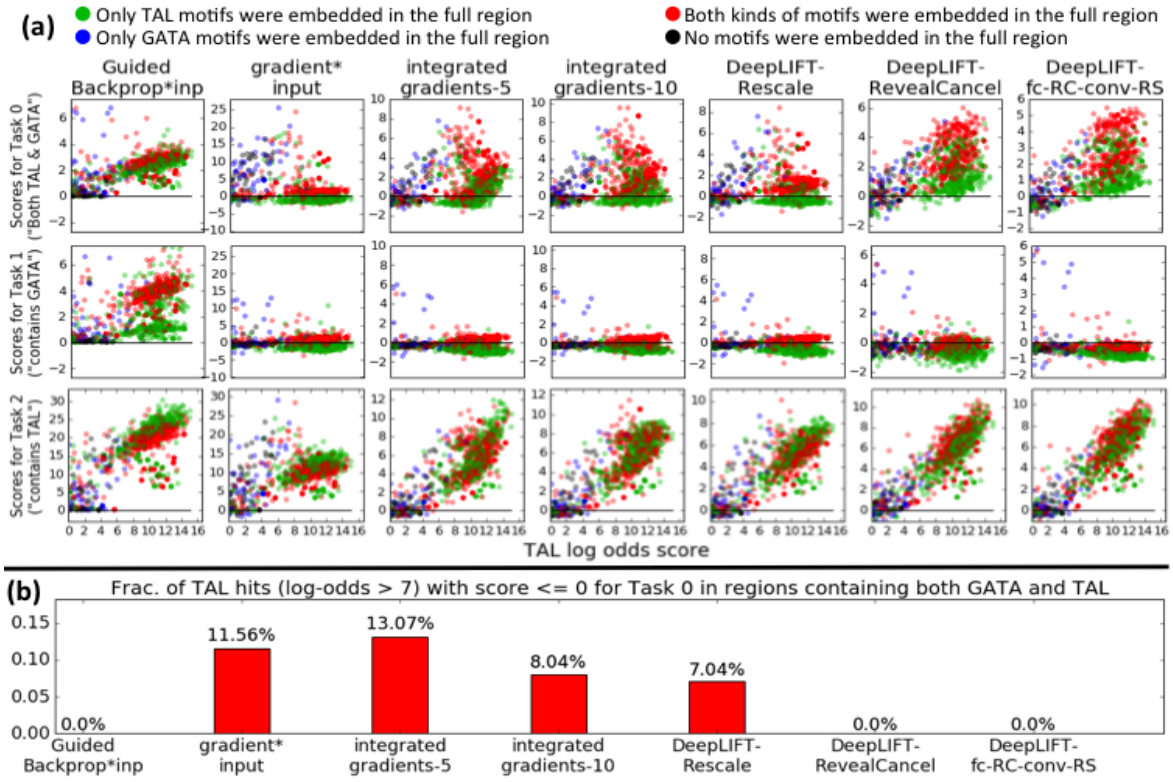


Figure 5. **DeepLIFT with RevealCancel gives qualitatively desirable behavior on TAL-GATA simulation**. (a) Scatter plots of importance score vs. strength of TAL1 motif match for different tasks and methods (see **Appendix G** for GATA1). For each region, top 5 motif matches are plotted. X-axes: log-odds of TAL1 motif match vs. background. Y-axes: total importance assigned to the match for specified task. Red dots are from regions where both TAL1 and GATA1 motifs were inserted during simulation; blue have GATA1 only, green have TAL1 only, black have no motifs inserted. "DeepLIFT-fc-RC-conv-RS" refers to using RevealCancel on the fully-connected layer and Rescale on the convolutional layers, which appears to reduce noise relative to using RevealCancel on all layers. (b) proportion of strong matches (log-odds > 7) to TAL1 motif in regions containing both TAL1 and GATA1 that had total score ≤ 0 for task 0; Guided Backprop$\times$inp and DeepLIFT with RevealCancel have no false negatives, but Guided Backprop has false positives for Task 1 (Panel (a))

4.2. Classifying Regulatory DNA (Genomics)

Next, we compared the importance scoring methods when applied to classification tasks on DNA sequence inputs (strings over the alphabet {A,C,G,T}). The human genome has millions of DNA sequence elements (200-1000 in length) containing specific combinations of short functional words to which regulatory proteins (RPs) bind to regulate gene activity. Each RP (e.g. GATA1) has binding affinity to specific collections of short DNA words (motifs) (e.g. GATAA and GATTA). A key problem in computational genomics is the discovery of motifs in regulatory DNA elements that give rise to distinct molecular signatures (labels) which can be measured experimentally. Here, in order to benchmark DeepLIFT and competing methods to uncover predictive patterns in DNA sequences, we design a simple simulation that captures the essence of the motif discovery problem described above.

Background DNA sequences of length 200 were generated by sampling the letters ACGT at each position with probabilities $0.3, 0.2, 0.2$ and 0.3 respectively. Motif instances were randomly sampled from previously known probabilistic motif models (See **Appendix F**) of two RPs named GATA1 and TAL1 (**Fig. 6a**)(Kheradpour & Kellis, 2014), and 0-3 instances of a given motif were inserted at random non-overlapping positions in the DNA sequences. We trained a multi-task neural network with two convolutional layers, global average pooling and one fully-connected layer on 3 binary classification tasks. Positive labeled sequences in task 1 represented "both GATA1 and TAL1 present", task 2 represented "GATA1 present" and in task 3 represented "TAL1 present". $\frac{1}{4}$ of sequences had both GATA1 and TAL1 motifs (labeled 111), $\frac{1}{4}$ had only GATA1 (labeled 010), $\frac{1}{4}$ had only TAL1 (labeled 001), and $\frac{1}{4}$ had no motifs (labeled 000). Details of the simulation, network architecture and predictive performance are given in **Appendix F**. For DeepLIFT and integrated gradients, we used a reference input that had the expected frequencies of ACGT at each position (i.e. we set the ACGT channel axis to $0.3, 0.2, 0.2, 0.3$; see **Appendix J** for results using

shuffled sequences as a reference). For fair comparison, this reference was also used for gradient×input and Guided Backprop×input ("input" is more accurately called Δinput where Δ measured w.r.t the reference). For DNA sequence inputs, we found Guided Backprop×input performed better than vanilla Guided Backprop; thus, we used the former.

Given a particular subsequence, it is possible to compute the log-odds score that the subsequence was sampled from a particular motif vs. originating from the background distribution of ACGT. To evaluate different importance-scoring methods, we found the top 5 matches (as ranked by their log-odds score) to each motif for each sequence from the test set, as well as the total importance allocated to the match by different importance-scoring methods for each task. The results are shown in **Fig. 5** (for TAL1) and **Appendix E** (for GATA1). Ideally, we expect an importance scoring method to show the following properties: (1) high scores for TAL1 motifs on task 2 and (2) low scores for TAL1 on task 1, with (3) higher scores corresponding to stronger log-odds matches; analogous pattern for GATA1 motifs (high for task 1, low for task 2); (4) high scores for both TAL1 and GATA1 motifs for task 0, with (5) higher scores on sequences containing both kinds of motifs vs. sequences containing only one kind (revealing cooperativity; corresponds to red dots lying above green dots in **Fig. 5**).

We observe Guided Backprop×input fails (2) by assigning positive importance to TAL1 on task 1 (see **Appendix H** for an example sequence). It fails property (4) by failing to identify cooperativity in task 0 (red dots overlay green dots). Both Guided Backprop×input and gradient×input show suboptimal behavior regarding property (3), in that there is a sudden increase in importance when the log-odds score is around 7, but little differentiation at higher log-odds scores (by contrast, the other methods show a more gradual increase). As a result, Guided Backprop×input and gradient×input can assign unduly high importance to weak motif matches (**Fig. 6**). This is a practical consequence of the thresholding problem from **Fig. 2**. The large discontinuous jumps in gradient also result in inflated scores (note the scale on the y-axes) relative to other methods.

We explored three versions of DeepLIFT: Rescale at all nonlinearities (DeepLIFT-Rescale), RevealCancel at all nonlinearities (DeepLIFT-RevealCancel), and Rescale at convolutional layers with RevealCancel at the fully connected layer (DeepLIFT-fc-RC-conv-RS). In contrast to the results on MNIST, we found that DeepLIFT-fc-RC-conv-RS reduced noise relative to pure RevealCancel. We think this is because of the noise-suppression property discussed in **Section 3.5.3**; if the convolutional layers act like motif detectors, the input to convolutional neurons that do not fire may just represent noise and importance should not be propagated to them (see **Fig. 6** for an example sequence).

Gradient×inp, integrated gradients and DeepLIFT-Rescale occasionally miss relevance of TAL1 for Task 0 (**Fig. 5b**), which is corrected by using RevealCancel on the fully connected layer (see example sequence in **Fig. 6**). Note that the RevealCancel scores seem to be tiered. As illustrated in **Appendix I**, this is related to having multiple instances of a given motif in a sequence (eg: when there are multiple TAL1 motifs, the importance assigned to the presence of TAL1 is distributed across all the motifs).

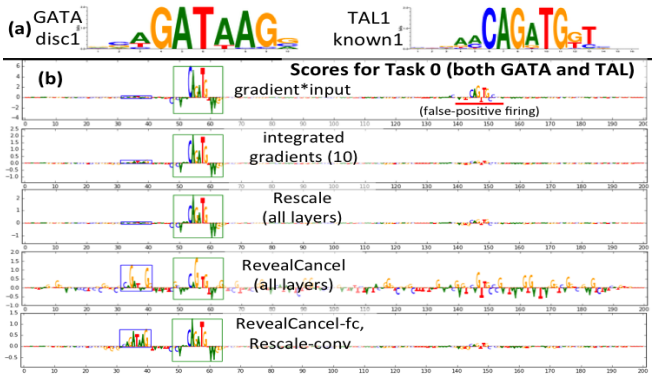


Figure 6. **RevealCancel highlights both TAL1 and GATA1 motifs for Task 0**. (a) PWM representations of the GATA1 motif and TAL1 motif used in the simulation (b) Scores for example sequence containing both TAL1 and GATA1 motifs. Letter height reflects the score. Blue box is location of embedded GATA1 motif, green box is location of embedded TAL1 motif. Red underline is chance occurrence of weak match to TAL1 (CAGTTG instead of CAGATG). Both TAL1 and GATA1 motifs should be highlighted for Task 0. RevealCancel on only the fully-connected layer reduces noise compared to RevealCancel on all layers.

5. Conclusion

We have presented DeepLIFT, a novel approach for computing importance scores based on explaining the difference of the output from some 'reference' output in terms of differences of the inputs from their 'reference' inputs. Using the difference-from-reference allows information to propagate even when the gradient is zero (**Fig. 1**), which could prove especially useful in Recurrent Neural Networks where saturating activations like sigmoid or tanh are popular. DeepLIFT avoids placing potentially misleading importance on bias terms (in contrast to gradient*input - see **Fig. 2**). By allowing separate treatment of positive and negative contributions, the DeepLIFT-RevealCancel rule can identify dependencies missed by other methods (**Fig. 3**). Open questions include how to apply DeepLIFT to RNNs, how to compute a good reference empirically from the data, and how best to propagate importance through 'max' operations (as in Maxout or Maxpooling neurons) beyond simply using the gradients.

6. Appendix

The appendix can be downloaded at:
http://proceedings.mlr.press/v70/shrikumar17a/shrikumar17a-supp.pdf

References

Bach, Sebastian, Binder, Alexander, Montavon, Grégoire, Klauschen, Frederick, Müller, Klaus-Robert, and Samek, Wojciech. On Pixel-Wise explanations for Non-Linear classifier decisions by Layer-Wise relevance propagation. *PLoS One*, 10(7):e0130140, 10 July 2015.

Chollet, Franois. keras. https://github.com/fchollet/keras, 2015.

Kheradpour, Pouya and Kellis, Manolis. Systematic discovery and characterization of regulatory motifs in encode tf binding experiments. *Nucleic acids research*, 42 (5):2976–2987, 2014.

Kindermans, Pieter-Jan, Schtt, Kristof, Mller, Klaus-Robert, and Dhne, Sven. Investigating the influence of noise and distractors on the interpretation of neural networks. *CoRR*, abs/1611.07270, 2016. URL https://arxiv.org/abs/1611.07270.

LeCun, Yann, Cortes, Corinna, and Burges, Christopher J.C. The mnist database of handwritten digits. http://yann.lecun.com/exdb/mnist/, 1999.

Lundberg, Scott and Lee, Su-In. An unexpected unity among methods for interpreting model predictions. *CoRR*, abs/1611.07478, 2016. URL http://arxiv.org/abs/1611.07478.

Selvaraju, Ramprasaath R., Das, Abhishek, Vedantam, Ramakrishna, Cogswell, Michael, Parikh, Devi, and Batra, Dhruv. Grad-cam: Why did you say that? visual explanations from deep networks via gradient-based localization. *CoRR*, abs/1610.02391, 2016. URL http://arxiv.org/abs/1610.02391.

Shrikumar, Avanti, Greenside, Peyton, Shcherbina, Anna, and Kundaje, Anshul. Not just a black box: Learning important features through propagating activation differences. *arXiv preprint arXiv:1605.01713*, 2016.

Simonyan, Karen, Vedaldi, Andrea, and Zisserman, Andrew. Deep inside convolutional networks: Visualising image classification models and saliency maps. *arXiv preprint arXiv:1312.6034*, 2013.

Springenberg, Jost Tobias, Dosovitskiy, Alexey, Brox, Thomas, and Riedmiller, Martin A. Striving for simplicity: The all convolutional net. *CoRR*, abs/1412.6806, 2014. URL http://arxiv.org/abs/1412.6806.

Sundararajan, Mukund, Taly, Ankur, and Yan, Qiqi. Gradients of counterfactuals. *CoRR*, abs/1611.02639, 2016. URL http://arxiv.org/abs/1611.02639.

Zeiler, Matthew D. and Fergus, Rob. Visualizing and understanding convolutional networks. *CoRR*, abs/1311.2901, 2013. URL http://arxiv.org/abs/1311.2901.

Zhou, Jian and Troyanskaya, Olga G. Predicting effects of noncoding variants with deep learning-based sequence model. *Nat Methods*, 12:931–4, 2015 Oct 2015. ISSN 1548-7105. doi: 10.1038/nmeth.3547.

Zintgraf, Luisa M, Cohen, Taco S, Adel, Tameem, and Welling, Max. Visualizing deep neural network decisions: Prediction difference analysis. *ICLR*, 2017. URL https://openreview.net/pdf?id=BJ5UeU9xx.

7. Acknowledgements

We thank Anna Shcherbina for early experiments applying DeepLIFT to image data and beta-testing. We thank Sinhan Kang of Korea University for identifying a typing error in Section 3.6.

8. Funding

AS was supported by a Howard Hughes Medical Institute International Student Research Fellowship and a Bio-X Bowes Fellowship. PG was supported by a Bio-X Stanford Interdisciplinary Graduate Fellowship. AK was supported by NIH grants DP2-GM-123485 and 1R01ES025009-02.

9. Author Contributions

AS & PG conceptualized DeepLIFT. AS implemented DeepLIFT. AS ran experiments on MNIST. AS & PG ran experiments on genomic data. AK provided guidance and feedback. AS, PG and AK wrote the manuscript.

Communication-Efficient Learning of Deep Networks from Decentralized Data

H. Brendan McMahan **Eider Moore** **Daniel Ramage** **Seth Hampson** **Blaise Agüera y Arcas**
Google, Inc., 651 N 34th St., Seattle, WA 98103 USA

arXiv:1602.05629v4 [cs.LG] 26 Jan 2023

Abstract

Modern mobile devices have access to a wealth of data suitable for learning models, which in turn can greatly improve the user experience on the device. For example, language models can improve speech recognition and text entry, and image models can automatically select good photos. However, this rich data is often privacy sensitive, large in quantity, or both, which may preclude logging to the data center and training there using conventional approaches. We advocate an alternative that leaves the training data distributed on the mobile devices, and learns a shared model by aggregating locally-computed updates. We term this decentralized approach *Federated Learning*.

We present a practical method for the federated learning of deep networks based on iterative model averaging, and conduct an extensive empirical evaluation, considering five different model architectures and four datasets. These experiments demonstrate the approach is robust to the unbalanced and non-IID data distributions that are a defining characteristic of this setting. Communication costs are the principal constraint, and we show a reduction in required communication rounds by 10–100× as compared to synchronized stochastic gradient descent.

1 Introduction

Increasingly, phones and tablets are the primary computing devices for many people [30, 2]. The powerful sensors on these devices (including cameras, microphones, and GPS), combined with the fact they are frequently carried, means they have access to an unprecedented amount of data, much of it private in nature. Models learned on such data hold the promise of greatly improving usability by powering more intelligent applications, but the sensitive nature of the data means there are risks and responsibilities to storing it in a centralized location.

We investigate a learning technique that allows users to collectively reap the benefits of shared models trained from this rich data, without the need to centrally store it. We term our approach *Federated Learning*, since the learning task is solved by a loose federation of participating devices (which we refer to as *clients*) which are coordinated by a central *server*. Each client has a local training dataset which is never uploaded to the server. Instead, each client computes an update to the current global model maintained by the server, and only this update is communicated. This is a direct application of the principle of *focused collection* or *data minimization* proposed by the 2012 White House report on privacy of consumer data [39]. Since these updates are specific to improving the current model, there is no reason to store them once they have been applied.

A principal advantage of this approach is the decoupling of model training from the need for direct access to the raw training data. Clearly, some trust of the server coordinating the training is still required. However, for applications where the training objective can be specified on the basis of data available on each client, federated learning can significantly reduce privacy and security risks by limiting the attack surface to only the device, rather than the device and the cloud.

Our primary contributions are 1) the identification of the problem of training on decentralized data from mobile devices as an important research direction; 2) the selection of a straightforward and practical algorithm that can be applied to this setting; and 3) an extensive empirical evaluation of the proposed approach. More concretely, we introduce the `FederatedAveraging` algorithm, which combines local stochastic gradient descent (SGD) on each client with a server that performs model averaging. We perform extensive experiments on this algorithm, demonstrating it is robust to unbalanced and non-IID data distributions, and can reduce the rounds of communication needed to train a deep network on decentralized data by orders of magnitude.

Appearing in Proceedings of the 20^{th} International Conference on Artificial Intelligence and Statistics (AISTATS) 2017, Fort Lauderdale, Flordia, USA. JMLR: W&CP volume 54. Copyright 2017 by the authors.

Federated Learning Ideal problems for federated learning have the following properties: 1) Training on real-world data from mobile devices provides a distinct advantage over training on proxy data that is generally available in the data center. 2) This data is privacy sensitive or large in size (compared to the size of the model), so it is preferable not to log it to the data center purely for the purpose of model training (in service of the *focused collection* principle). 3) For supervised tasks, labels on the data can be inferred naturally from user interaction.

Many models that power intelligent behavior on mobile devices fit the above criteria. As two examples, we consider *image classification*, for example predicting which photos are most likely to be viewed multiple times in the future, or shared; and *language models*, which can be used to improve voice recognition and text entry on touch-screen keyboards by improving decoding, next-word-prediction, and even predicting whole replies [10]. The potential training data for both these tasks (all the photos a user takes and everything they type on their mobile keyboard, including passwords, URLs, messages, etc.) can be privacy sensitive. The distributions from which these examples are drawn are also likely to differ substantially from easily available proxy datasets: the use of language in chat and text messages is generally much different than standard language corpora, e.g., Wikipedia and other web documents; the photos people take on their phone are likely quite different than typical Flickr photos. And finally, the labels for these problems are directly available: entered text is self-labeled for learning a language model, and photo labels can be defined by natural user interaction with their photo app (which photos are deleted, shared, or viewed).

Both of these tasks are well-suited to learning a neural network. For image classification feed-forward deep networks, and in particular convolutional networks, are well-known to provide state-of-the-art results [26, 25]. For language modeling tasks recurrent neural networks, and in particular LSTMs, have achieved state-of-the-art results [20, 5, 22].

Privacy Federated learning has distinct privacy advantages compared to data center training on persisted data. Holding even an "anonymized" dataset can still put user privacy at risk via joins with other data [37]. In contrast, the information transmitted for federated learning is the minimal update necessary to improve a particular model (naturally, the strength of the privacy benefit depends on the content of the updates).[1] The updates themselves can (and should) be ephemeral. They will never contain more information than the raw training data (by the data processing inequality), and will generally contain much less. Further, the source of the updates is not needed by the aggregation algorithm, so updates can be transmitted without identifying meta-data over a mix network such as Tor [7] or via a trusted third party. We briefly discuss the possibility of combining federated learning with secure multiparty computation and differential privacy at the end of the paper.

[1] For example, if the update is the total gradient of the loss on all of the local data, and the features are a sparse bag-of-words, then the non-zero gradients reveal exactly which words the user has entered on the device. In contrast, the sum of many gradients for a dense model such as a CNN offers a harder target for attackers seeking information about individual training instances (though attacks are still possible).

Federated Optimization We refer to the optimization problem implicit in federated learning as federated optimization, drawing a connection (and contrast) to distributed optimization. Federated optimization has several key properties that differentiate it from a typical distributed optimization problem:

- **Non-IID** The training data on a given client is typically based on the usage of the mobile device by a particular user, and hence any particular user's local dataset will not be representative of the population distribution.
- **Unbalanced** Similarly, some users will make much heavier use of the service or app than others, leading to varying amounts of local training data.
- **Massively distributed** We expect the number of clients participating in an optimization to be much larger than the average number of examples per client.
- **Limited communication** Mobile devices are frequently offline or on slow or expensive connections.

In this work, our emphasis is on the non-IID and unbalanced properties of the optimization, as well as the critical nature of the communication constraints. A deployed federated optimization system must also address a myriad of practical issues: client datasets that change as data is added and deleted; client availability that correlates with the local data distribution in complex ways (e.g., phones from speakers of American English will likely be plugged in at different times than speakers of British English); and clients that never respond or send corrupted updates.

These issues are beyond the scope of the current work; instead, we use a controlled environment that is suitable for experiments, but still addresses the key issues of client availability and unbalanced and non-IID data. We assume a synchronous update scheme that proceeds in rounds of communication. There is a fixed set of K clients, each with a fixed local dataset. At the beginning of each round, a random fraction C of clients is selected, and the server sends the current global algorithm state to each of these clients (e.g., the current model parameters). We only select a fraction of clients for efficiency, as our experiments show diminishing returns for adding more clients beyond a certain point. Each selected client then performs local computation based on the global state and its local dataset, and sends an update to the server. The server then applies these updates to its global state, and the process repeats.

While we focus on non-convex neural network objectives, the algorithm we consider is applicable to any finite-sum objective of the form

$$\min_{w \in \mathbb{R}^d} f(w) \qquad \text{where} \qquad f(w) \stackrel{\text{def}}{=} \frac{1}{n} \sum_{i=1}^{n} f_i(w). \tag{1}$$

For a machine learning problem, we typically take $f_i(w) = \ell(x_i, y_i; w)$, that is, the loss of the prediction on example (x_i, y_i) made with model parameters w. We assume there are K clients over which the data is partitioned, with $\mathcal{P}_k$ the set of indexes of data points on client k, with $n_k = |\mathcal{P}_k|$. Thus, we can re-write the objective (1) as

$$f(w) = \sum_{k=1}^{K} \frac{n_k}{n} F_k(w) \quad \text{where} \quad F_k(w) = \frac{1}{n_k} \sum_{i \in \mathcal{P}_k} f_i(w).$$

If the partition $\mathcal{P}_k$ was formed by distributing the training examples over the clients uniformly at random, then we would have $\mathbb{E}_{\mathcal{P}_k}[F_k(w)] = f(w)$, where the expectation is over the set of examples assigned to a fixed client k. This is the IID assumption typically made by distributed optimization algorithms; we refer to the case where this does not hold (that is, F_k could be an arbitrarily bad approximation to f) as the non-IID setting.

In data center optimization, communication costs are relatively small, and computational costs dominate, with much of the recent emphasis being on using GPUs to lower these costs. In contrast, in federated optimization communication costs dominate — we will typically be limited by an upload bandwidth of 1 MB/s or less. Further, clients will typically only volunteer to participate in the optimization when they are charged, plugged-in, and on an unmetered wi-fi connection. Further, we expect each client will only participate in a small number of update rounds per day. On the other hand, since any single on-device dataset is small compared to the total dataset size, and modern smartphones have relatively fast processors (including GPUs), computation becomes essentially free compared to communication costs for many model types. Thus, our goal is to use additional computation in order to decrease the number of rounds of communication needed to train a model. There are two primary ways we can add computation: 1) *increased parallelism*, where we use more clients working independently between each communication round; and, 2) *increased computation on each client*, where rather than performing a simple computation like a gradient calculation, each client performs a more complex calculation between each communication round. We investigate both of these approaches, but the speedups we achieve are due primarily to adding more computation on each client, once a minimum level of parallelism over clients is used.

Related Work Distributed training by iteratively averaging locally trained models has been studied by McDonald et al. [28] for the perceptron and Povey et al. [31] for speech recognition DNNs. Zhang et al. [42] studies an asynchronous approach with "soft" averaging. These works only consider the cluster / data center setting (at most 16 workers, wall-clock time based on fast networks), and do not consider datasets that are unbalanced and non-IID, properties that are essential to the federated learning setting. We adapt this style of algorithm to the federated setting and perform the appropriate empirical evaluation, which asks different questions than those relevant in the data center setting, and requires different methodology.

Using similar motivation to ours, Neverova et al. [29] also discusses the advantages of keeping sensitive user data on device. The work of Shokri and Shmatikov [35] is related in several ways: they focus on training deep networks, emphasize the importance of privacy, and address communication costs by only sharing a subset of the parameters during each round of communication; however, they also do not consider unbalanced and non-IID data, and the empirical evaluation is limited.

In the convex setting, the problem of distributed optimization and estimation has received significant attention [4, 15, 33], and some algorithms do focus specifically on communication efficiency [45, 34, 40, 27, 43]. In addition to assuming convexity, this existing work generally requires that the number of clients is much smaller than the number of examples per client, that the data is distributed across the clients in IID fashion, and that each node has an identical number of data points — all of these assumptions are violated in the federated optimization setting. Asynchronous distributed forms of SGD have also been applied to training neural networks, e.g., Dean et al. [12], but these approaches require a prohibitive number of updates in the federated setting. Distributed consensus algorithms (e.g., [41]) relax the IID assumption, but are still not a good fit for communication-constrained optimization over very many clients.

One endpoint of the (parameterized) algorithm family we consider is simple one-shot averaging, where each client solves for the model that minimizes (possibly regularized) loss on their local data, and these models are averaged to produce the final global model. This approach has been studied extensively in the convex case with IID data, and it is known that in the worst-case, the global model produced is no better than training a model on a single client [44, 3, 46].

2 The `FederatedAveraging` Algorithm

The recent multitude of successful applications of deep learning have almost exclusively relied on variants of stochastic gradient descent (SGD) for optimization; in fact, many advances can be understood as adapting the structure of the model (and hence the loss function) to be more amenable to optimization by simple gradient-based methods [16]. Thus, it is natural that we build algorithms for

federated optimization by starting from SGD.

SGD can be applied naively to the federated optimization problem, where a single batch gradient calculation (say on a randomly selected client) is done per round of communication. This approach is computationally efficient, but requires very large numbers of rounds of training to produce good models (e.g., even using an advanced approach like batch normalization, Ioffe and Szegedy [21] trained MNIST for 50000 steps on minibatches of size 60). We consider this baseline in our CIFAR-10 experiments.

In the federated setting, there is little cost in wall-clock time to involving more clients, and so for our baseline we use large-batch synchronous SGD; experiments by Chen et al. [8] show this approach is state-of-the-art in the data center setting, where it outperforms asynchronous approaches. To apply this approach in the federated setting, we select a C-fraction of clients on each round, and compute the gradient of the loss over all the data held by these clients. Thus, C controls the *global* batch size, with $C = 1$ corresponding to full-batch (non-stochastic) gradient descent.[2] We refer to this baseline algorithm as `FederatedSGD` (or `FedSGD`).

A typical implementation of `FedSGD` with $C = 1$ and a fixed learning rate η has each client k compute $g_k = \nabla F_k(w_t)$, the average gradient on its local data at the current model w_t, and the central server aggregates these gradients and applies the update $w_{t+1} \leftarrow w_t - \eta \sum_{k=1}^{K} \frac{n_k}{n} g_k$, since $\sum_{k=1}^{K} \frac{n_k}{n} g_k = \nabla f(w_t)$. An equivalent update is given by $\forall k, w_{t+1}^k \leftarrow w_t - \eta g_k$ and then $w_{t+1} \leftarrow \sum_{k=1}^{K} \frac{n_k}{n} w_{t+1}^k$. That is, each client locally takes one step of gradient descent on the current model using its local data, and the server then takes a weighted average of the resulting models. Once the algorithm is written this way, we can add more computation to each client by iterating the local update $w^k \leftarrow w^k - \eta \nabla F_k(w^k)$ multiple times before the averaging step. We term this approach `FederatedAveraging` (or `FedAvg`). The amount of computation is controlled by three key parameters: C, the fraction of clients that perform computation on each round; E, then number of training passes each client makes over its local dataset on each round; and B, the local minibatch size used for the client updates. We write $B = \infty$ to indicate that the full local dataset is treated as a single minibatch. Thus, at one endpoint of this algorithm family, we can take $B = \infty$ and $E = 1$ which corresponds exactly to `FedSGD`. For a client with n_k local examples, the number of local updates per round is given by $u_k = E\frac{n_k}{B}$; Complete pseudo-code is given in Algorithm 1.

For general non-convex objectives, averaging models in parameter space could produce an arbitrarily bad model.

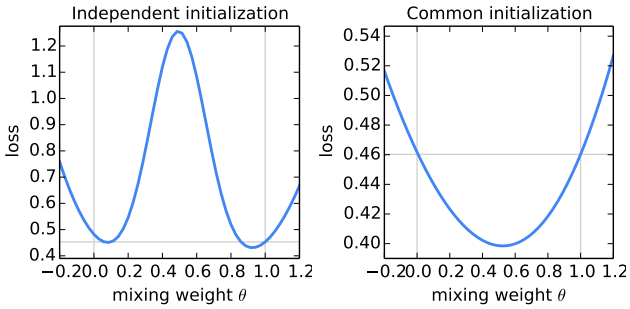


Figure 1: The loss on the full MNIST training set for models generated by averaging the parameters of two models w and w' using $\theta w + (1-\theta)w'$ for 50 evenly spaced values $\theta \in [-0.2, 1.2]$.The models w and w' were trained using SGD on different small datasets. For the left plot, w and w' were initialized using different random seeds; for the right plot, a shared seed was used. Note the different y-axis scales. The horizontal line gives the best loss achieved by w or w' (which were quite close, corresponding to the vertical lines at $\theta = 0$ and $\theta = 1$). With shared initialization, averaging the models produces a significant reduction in the loss on the total training set (much better than the loss of either parent model).

Following the approach of Goodfellow et al. [17], we see exactly this bad behavior when we average two MNIST digit-recognition models[3] trained from different initial conditions (Figure 1, left). For this figure, the parent models w and w' were each trained on non-overlapping IID samples of 600 examples from the MNIST training set. Training was via SGD with a fixed learning rate of 0.1 for 240 updates on minibatches of size 50 (or $E = 20$ passes over the mini-datasets of size 600). This is approximately the amount of training where the models begin to overfit their local datasets.

Recent work indicates that in practice, the loss surfaces of sufficiently over-parameterized NNs are surprisingly well-behaved and in particular less prone to bad local minima than previously thought [11, 17, 9]. And indeed, when we start two models *from the same random initialization* and then again train each independently on a different subset of the data (as described above), we find that naive parameter averaging works surprisingly well (Figure 1, right): the average of these two models, $\frac{1}{2}w + \frac{1}{2}w'$, achieves significantly lower loss on the full MNIST training set than the best model achieved by training on either of the small datasets independently. While Figure 1 starts from a random initialization, note a shared starting model w_t is used for each round of `FedAvg`, and so the same intuition applies.

[2] While the batch selection mechanism is different than selecting a batch by choosing individual examples uniformly at random, the batch gradients g computed by `FedSGD` still satisfy $\mathbb{E}[g] = \nabla f(w)$.

[3] We use the "2NN" multi-layer perceptron described in Section 3.

Algorithm 1 `FederatedAveraging`. The K clients are indexed by k; B is the local minibatch size, E is the number of local epochs, and η is the learning rate.

Server executes:
initialize w_0
for each round $t = 1, 2, \dots$ **do**
  $m \leftarrow \max(C \cdot K, 1)$
  $S_t \leftarrow$ (random set of m clients)
  for each client $k \in S_t$ **in parallel do**
    $w_{t+1}^k \leftarrow \text{ClientUpdate}(k, w_t)$
  $m_t \leftarrow \sum_{k \in S_t} n_k$
  $w_{t+1} \leftarrow \sum_{k \in S_t} \frac{n_k}{m_t} w_{t+1}^k$ // *Erratum*[4]

ClientUpdate(k, w): // *Run on client* k
$\mathcal{B} \leftarrow$ (split $\mathcal{P}_k$ into batches of size B)
for each local epoch i from 1 to E **do**
  for batch $b \in \mathcal{B}$ **do**
    $w \leftarrow w - \eta \nabla \ell(w; b)$
return w to server

3 Experimental Results

We are motivated by both image classification and language modeling tasks where good models can greatly enhance the usability of mobile devices. For each of these tasks we first picked a proxy dataset of modest enough size that we could thoroughly investigate the hyperparameters of the `FedAvg` algorithm. While each individual training run is relatively small, we trained over 2000 individual models for these experiments. We then present results on the benchmark CIFAR-10 image classification task. Finally, to demonstrate the effectiveness of `FedAvg` on a real-world problem with a natural partitioning of the data over clients, we evaluate on a large language modeling task.

Our initial study includes three model families on two datasets. The first two are for the MNIST digit recognition task [26]: 1) A simple multilayer-perceptron with 2-hidden layers with 200 units each using ReLu activations (199,210 total parameters), which we refer to as the MNIST 2NN. 2) A CNN with two 5x5 convolution layers (the first with 32 channels, the second with 64, each followed with 2x2 max pooling), a fully connected layer with 512 units and ReLu activation, and a final softmax output layer (1,663,370 total parameters). To study federated optimization, we also need to specify how the data is distributed over the clients. We study two ways of partitioning the MNIST data over clients: **IID**, where the data is shuffled, and then partitioned into 100 clients each receiving 600 examples, and **Non-IID**, where we first *sort* the data by digit label, divide it into 200 shards of size 300, and assign each of 100 clients 2 shards. This is a pathological non-IID partition of the data, as most clients will only have examples of two digits, letting us explore the degree to which our algorithms will break on highly non-IID data. Both of these partitions are balanced, however.[5]

For language modeling, we built a dataset from *The Complete Works of William Shakespeare* [32]. We construct a client dataset for each speaking role in each play with at least two lines. This produced a dataset with 1146 clients. For each client, we split the data into a set of training lines (the first 80% of lines for the role), and test lines (the last 20%, rounded up to at least one line). The resulting dataset has 3,564,579 characters in the training set, and 870,014 characters[6] in the test set. This data is substantially unbalanced, with many roles having only a few lines, and a few with a large number of lines. Further, observe the test set is not a random sample of lines, but is temporally separated by the chronology of each play. Using an identical train/test split, we also form a balanced and IID version of the dataset, also with 1146 clients.

On this data we train a stacked character-level LSTM language model, which after reading each character in a line, predicts the next character [22]. The model takes a series of characters as input and embeds each of these into a learned 8 dimensional space. The embedded characters are then processed through 2 LSTM layers, each with 256 nodes. Finally the output of the second LSTM layer is sent to a softmax output layer with one node per character. The full model has 866,578 parameters, and we trained using an unroll length of 80 characters.

SGD is sensitive to the tuning of the learning-rate parameter η. The results reported here are based on training over a sufficiently wide grid of learning rates (typically 11-13 values for η on a multiplicative grid of resolution $10^{\frac{1}{3}}$ or $10^{\frac{1}{6}}$). We checked to ensure the best learning rates were in the middle of our grids, and that there was not a significant difference between the best learning rates. Unless otherwise noted, we plot metrics for the best performing rate selected individually for each x-axis value. We find that the optimal learning rates do not vary too much as a function of the other parameters.

Increasing parallelism We first experiment with the client fraction C, which controls the amount of multi-client parallelism. Table 1 shows the impact of varying C for both MNIST models. We report the number of communication rounds necessary to achieve a target test-set accuracy. To compute this, we construct a learning curve for each combination of parameter settings, optimizing η as described above and then making each curve monotonically improving by taking the best value of test-set accuracy achieved over

[4] Earlier versions of this paper incorrectly indicated summation over all K clients here.

[5] We performed additional experiments on unbalanced versions of these datasets, and found them to in fact be slightly easier for `FedAvg`.

[6] We always use character to refer to a one byte string, and use role to refer to a part in the play.

Table 1: Effect of the client fraction C on the MNIST 2NN with $E = 1$ and CNN with $E = 5$. Note $C = 0.0$ corresponds to one client per round; since we use 100 clients for the MNIST data, the rows correspond to 1, 10 20, 50, and 100 clients. Each table entry gives the number of rounds of communication necessary to achieve a test-set accuracy of 97% for the 2NN and 99% for the CNN, along with the speedup relative to the $C = 0$ baseline. Five runs with the large batch size did not reach the target accuracy in the allowed time.

2NN	IID		NON-IID	
C	$B = \infty$	$B = 10$	$B = \infty$	$B = 10$
0.0	1455	316	4278	3275
0.1	1474 (1.0×)	87 (3.6×)	1796 (2.4×)	664 (4.9×)
0.2	1658 (0.9×)	77 (4.1×)	1528 (2.8×)	619 (5.3×)
0.5	— (—)	75 (4.2×)	— (—)	443 (7.4×)
1.0	— (—)	70 (4.5×)	— (—)	380 (8.6×)
CNN, $E = 5$				
0.0	387	50	1181	956
0.1	339 (1.1×)	18 (2.8×)	1100 (1.1×)	206 (4.6×)
0.2	337 (1.1×)	18 (2.8×)	978 (1.2×)	200 (4.8×)
0.5	164 (2.4×)	18 (2.8×)	1067 (1.1×)	261 (3.7×)
1.0	246 (1.6×)	16 (3.1×)	— (—)	97 (9.9×)

all prior rounds. We then calculate the number of rounds where the curve crosses the target accuracy, using linear interpolation between the discrete points forming the curve. This is perhaps best understood by reference to Figure 2, where the gray lines show the targets.

With $B = \infty$ (for MNIST processing all 600 client examples as a single batch per round), there is only a small advantage in increasing the client fraction. Using the smaller batch size $B = 10$ shows a significant improvement in using $C \geq 0.1$, especially in the non-IID case. Based on these results, for most of the remainder of our experiments we fix $C = 0.1$, which strikes a good balance between computational efficiency and convergence rate. Comparing the number of rounds for the $B = \infty$ and $B = 10$ columns in Table 1 shows a dramatic speedup, which we investigate next.

Increasing computation per client In this section, we fix $C = 0.1$, and add more computation per client on each round, either decreasing B, increasing E, or both. Figure 2 demonstrates that adding more local SGD updates per round can produce a dramatic decrease in communication costs, and Table 2 quantifies these speedups. The expected number of updates per client per round is $u = (\mathbb{E}[n_k]/B)E = nE/(KB)$, where the expectation is over the draw of a random client k. We order the rows in each section of Table 2 by this statistic. We see that increasing u by varying both E and B is effective. As long as B is large enough to take full advantage of available parallelism on the client hardware, there is essentially no cost in computation time for lowering it, and so in practice this should be the first parameter tuned.

Table 2: Number of communication rounds to reach a target accuracy for `FedAvg`, versus `FedSGD` (first row, $E = 1$ and $B = \infty$). The u column gives $u = En/(KB)$, the expected number of updates per round.

MNIST CNN, 99% ACCURACY					
CNN	E	B	u	IID	NON-IID
FEDSGD	1	∞	1	626	483
FEDAVG	5	∞	5	179 (3.5×)	1000 (0.5×)
FEDAVG	1	50	12	65 (9.6×)	600 (0.8×)
FEDAVG	20	∞	20	234 (2.7×)	672 (0.7×)
FEDAVG	1	10	60	34 (18.4×)	350 (1.4×)
FEDAVG	5	50	60	29 (21.6×)	334 (1.4×)
FEDAVG	20	50	240	32 (19.6×)	426 (1.1×)
FEDAVG	5	10	300	20 (31.3×)	229 (2.1×)
FEDAVG	20	10	1200	18 (34.8×)	173 (2.8×)
SHAKESPEARE LSTM, 54% ACCURACY					
LSTM	E	B	u	IID	NON-IID
FEDSGD	1	∞	1.0	2488	3906
FEDAVG	1	50	1.5	1635 (1.5×)	549 (7.1×)
FEDAVG	5	∞	5.0	613 (4.1×)	597 (6.5×)
FEDAVG	1	10	7.4	460 (5.4×)	164 (23.8×)
FEDAVG	5	50	7.4	401 (6.2×)	152 (25.7×)
FEDAVG	5	10	37.1	192 (13.0×)	41 (95.3×)

For the IID partition of the MNIST data, using more computation per client decreases the number of rounds to reach the target accuracy by $35\times$ for the CNN and $46\times$ for the 2NN (see Table 4 in Appendix A for details for the 2NN). The speedups for the pathologically partitioned non-IID data are smaller, but still substantial ($2.8 - 3.7\times$). It is impressive that averaging provides *any* advantage (vs. actually diverging) when we naively average the parameters of models trained on entirely different pairs of digits. Thus, we view this as strong evidence for the robustness of this approach.

The unbalanced and non-IID distribution of the Shakespeare (by role in the play) is much more representative of the kind of data distribution we expect for real-world applications. Encouragingly, for this problem learning on the non-IID and unbalanced data is actually much easier (a $95\times$ speedup vs $13\times$ for the balanced IID data); we conjecture this is largely due to the fact some roles have relatively large local datasets, which makes increased local training particularly valuable.

For all three model classes, `FedAvg` converges to a higher level of test-set accuracy than the baseline `FedSGD` models. This trend continues even if the lines are extended beyond the plotted ranges. For example, for the CNN the $B = \infty, E = 1$ `FedSGD` model eventually reaches 99.22% accuracy after 1200 rounds (and had not improved further after 6000 rounds), while the $B = 10, E = 20$ `FedAvg` model reaches an accuracy of 99.44% after 300 rounds. We conjecture that in addition to lowering communication costs, model averaging produces a regularization benefit similar to that achieved by dropout [36].

We are primarily concerned with generalization performance, but `FedAvg` is effective at optimizing the training loss as well, even beyond the point where test-set accuracy plateaus. We observed similar behavior for all three model classes, and present plots for the MNIST CNN in Figure 6

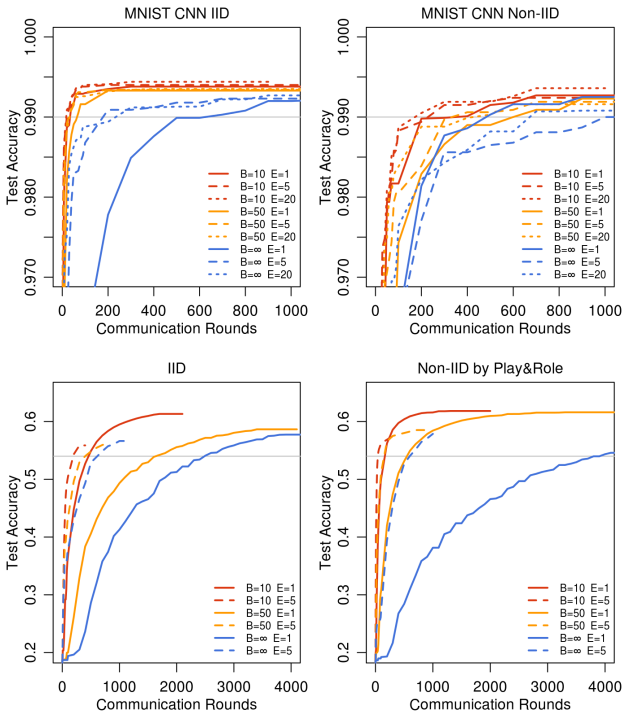


Figure 2: Test set accuracy vs. communication rounds for the MNIST CNN (IID and then pathological non-IID) and Shakespeare LSTM (IID and then by Play&Role) with $C = 0.1$ and optimized η. The gray lines show the target accuracies used in Table 2. Plots for the 2NN are given as Figure 7 in Appendix A.

in Appendix A.

Can we over-optimize on the client datasets? The current model parameters only influence the optimization performed in each `ClientUpdate` via initialization. Thus, as $E \to \infty$, at least for a convex problem eventually the initial conditions should be irrelevant, and the global minimum would be reached regardless of initialization. Even for a non-convex problem, one might conjecture the algorithm would converge to the same local minimum as long as the initialization was in the same basin. That is, we would expect that while one round of averaging might produce a reasonable model, additional rounds of communication (and averaging) would not produce further improvements.

Figure 3 shows the impact of large E during initial training on the Shakespeare LSTM problem. Indeed, for very large numbers of local epochs, `FedAvg` can plateau or diverge.[7] This result suggests that for some models, especially in the later stages of convergence, it may be useful to decay the

[7]Note that due to this behavior and because for large E not all experiments for all learning rates were run for the full number of rounds, we report results for a fixed learning rate (which perhaps surprisingly was near-optimal across the range of E parameters) and without forcing the lines to be monotonic.

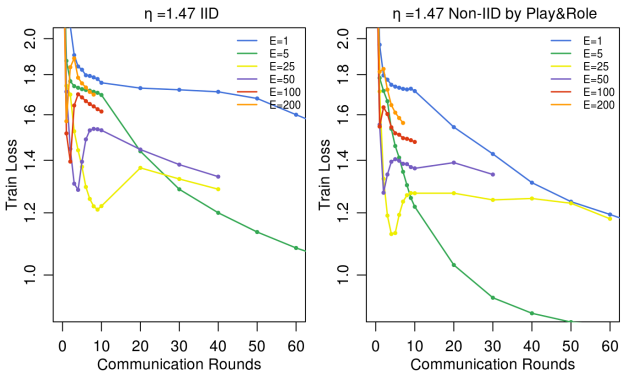


Figure 3: The effect of training for many local epochs (large E) between averaging steps, fixing $B = 10$ and $C = 0.1$ for the Shakespeare LSTM with a fixed learning rate $\eta = 1.47$.

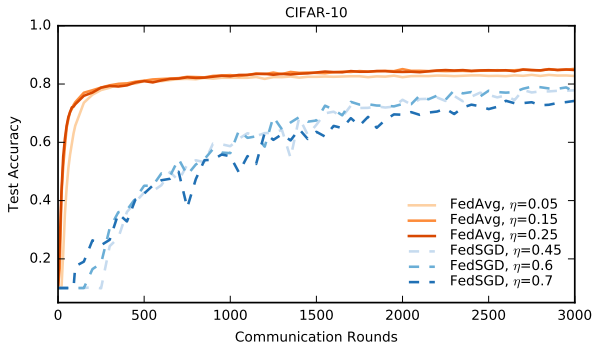


Figure 4: Test accuracy versus communication for the CIFAR10 experiments. `FedSGD` uses a learning-rate decay of 0.9934 per round; `FedAvg` uses $B = 50$, learning-rate decay of 0.99 per round, and $E = 5$.

amount of local computation per round (moving to smaller E or larger B) in the same way decaying learning rates can be useful. Figure 8 in Appendix A gives the analogous experiment for the MNIST CNN. Interestingly, for this model we see no significant degradation in the convergence rate for large values of E. However, we see slightly better performance for $E = 1$ versus $E = 5$ for the large-scale language modeling task described below (see Figure 10 in Appendix A).

CIFAR experiments We also ran experiments on the CIFAR-10 dataset [24] to further validate `FedAvg`. The dataset consists of 10 classes of 32x32 images with three RGB channels. There are 50,000 training examples and 10,000 testing examples, which we partitioned into 100 clients each containing 500 training and 100 testing examples; since there isn't a natural user partitioning of this data, we considered the balanced and IID setting. The model architecture was taken from the TensorFlow tutorial [38], which consists of two convolutional layers followed by two fully connected layers and then a linear transformation layer

Table 3: Number of rounds and speedup relative to baseline SGD to reach a target test-set accuracy on CIFAR10. SGD used a minibatch size of 100. FedSGD and FedAvg used $C = 0.1$, with FedAvg using $E = 5$ and $B = 50$.

ACC.	80%	82%	85%
SGD	18000 (—)	31000 (—)	99000 (—)
FEDSGD	3750 (4.8×)	6600 (4.7×)	N/A (—)
FEDAVG	280 (64.3×)	630 (49.2×)	2000 (49.5×)

to produce logits, for a total of about 10^6 parameters. Note that state-of-the-art approaches have achieved a test accuracy of 96.5% [19] for CIFAR; nevertheless, the standard model we use is sufficient for our needs, as our goal is to evaluate our optimization method, not achieve the best possible accuracy on this task. The images are preprocessed as part of the training input pipeline, which consists of cropping the images to 24x24, randomly flipping left-right and adjusting the contrast, brightness and whitening.

For these experiments, we considered an additional baseline, standard SGD training on the full training set (no user partitioning), using minibatches of size 100. We achieved an 86% test accuracy after 197,500 minibatch updates (each minibatch update requires a communication round in the federated setting). FedAvg achieves a similar test accuracy of 85% after only 2,000 communication rounds. For all algorithms, we tuned a learning-rate decay parameter in addition to the initial learning rate. Table 3 gives the number of communication rounds for baseline SGD, FedSGD, and FedAvg to reach three different accuracy targets, and Figure 4 gives learning-rate curves for FedAvg versus FedSGD.

By running experiments with minibatches of size $B = 50$ for both SGD and FedAvg, we can also look at accuracy as a function of the number of such minibatch gradient calculations. We expect SGD to do better here, because a sequential step is taken after each minibatch computation. However, as Figure 9 in the appendix shows, for modest values of C and E, FedAvg makes a similar amount of progress per minibatch computation. Further, we see that both standard SGD and FedAvg with only one client per round ($C = 0$), demonstrate significant oscillations in accuracy, whereas averaging over more clients smooths this out.

Large-scale LSTM experiments We ran experiments on a large-scale next-word prediction task to demonstrate the effectiveness of our approach on a real-world problem. Our training dataset consists 10 million public posts from a large social network. We grouped the posts by author, for a total of over 500,000 clients. This dataset is a realistic proxy for the type of text entry data that would be present on a user's mobile device. We limited each client dataset to at most 5000 words, and report accuracy (the fraction of the data where the highest predicted probability was on the correct next word, out of 10000 possibilities) on a test set of $1e5$

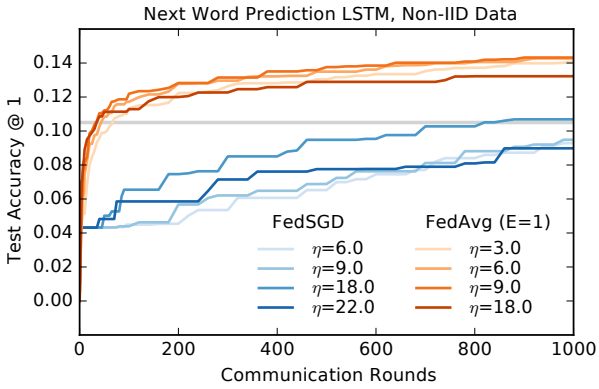


Figure 5: Monotonic learning curves for the large-scale language model word LSTM.

posts from different (non-training) authors. Our model is a 256 node LSTM on a vocabulary of 10,000 words. The input and output embeddings for each word were of dimension 192, and co-trained with the model; there are 4,950,544 parameters in all. We used an unroll of 10 words.

These experiments required significant computational resources and so we did not explore hyper-parameters as thoroughly: all runs trained on 200 clients per round; FedAvg used $B = 8$ and $E = 1$. We explored a variety of learning rates for FedAvg and the baseline FedSGD. Figure 5 shows monotonic learning curves for the best learning rates. FedSGD with $\eta = 18.0$ required 820 rounds to reach 10.5% accuracy, while FedAvg with $\eta = 9.0$ reached an accuracy of 10.5% in only 35 communication rounds (23× fewer then FedSGD). We observed lower variance in test accuracy for FedAvg, see Figure 10 in Appendix A. This figure also include results for $E = 5$, which performed slightly worse than $E = 1$.

4 Conclusions and Future Work

Our experiments show that federated learning can be made practical, as FedAvg trains high-quality models using relatively few rounds of communication, as demonstrated by results on a variety of model architectures: a multi-layer perceptron, two different convolutional NNs, a two-layer character LSTM, and a large-scale word-level LSTM.

While federated learning offers many practical privacy benefits, providing stronger guarantees via differential privacy [14, 13, 1], secure multi-party computation [18], or their combination is an interesting direction for future work. Note that both classes of techniques apply most naturally to synchronous algorithms like FedAvg.[8]

[8] Subsequent to this work, Bonawitz et al. [6] introduced an efficient secure aggregation protocol for federated learning, and Konečný et al. [23] presented algorithms for further decreasing communication costs.

References

[1] Martin Abadi, Andy Chu, Ian Goodfellow, Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In *23rd ACM Conference on Computer and Communications Security (ACM CCS)*, 2016.

[2] Monica Anderson. Technology device ownership: 2015. http://www.pewinternet.org/2015/10/29/technology-device-ownership-2015/, 2015.

[3] Yossi Arjevani and Ohad Shamir. Communication complexity of distributed convex learning and optimization. In *Advances in Neural Information Processing Systems 28*. 2015.

[4] Maria-Florina Balcan, Avrim Blum, Shai Fine, and Yishay Mansour. Distributed learning, communication complexity and privacy. *arXiv preprint arXiv:1204.3514*, 2012.

[5] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A neural probabilistic language model. *J. Mach. Learn. Res.*, 2003.

[6] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for federated learning on user-held data. In *NIPS Workshop on Private Multi-Party Machine Learning*, 2016.

[7] David L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM*, 24(2), 1981.

[8] Jianmin Chen, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. Revisiting distributed synchronous sgd. In *ICLR Workshop Track*, 2016.

[9] Anna Choromanska, Mikael Henaff, Michaël Mathieu, Gérard Ben Arous, and Yann LeCun. The loss surfaces of multilayer networks. In *AISTATS*, 2015.

[10] Greg Corrado. Computer, respond to this email. http://googleresearch.blogspot.com/2015/11/computer-respond-to-this-email.html, November 2015.

[11] Yann N. Dauphin, Razvan Pascanu, Çaglar Gülçehre, KyungHyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *NIPS*, 2014.

[12] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In *NIPS*, 2012.

[13] John Duchi, Michael I. Jordan, and Martin J. Wainwright. Privacy aware learning. *Journal of the Association for Computing Machinery*, 2014.

[14] Cynthia Dwork and Aaron Roth. *The Algorithmic Foundations of Differential Privacy*. Foundations and Trends in Theoretical Computer Science. Now Publishers, 2014.

[15] Olivier Fercoq, Zheng Qu, Peter Richtárik, and Martin Takác. Fast distributed coordinate descent for non-strongly convex losses. In *Machine Learning for Signal Processing (MLSP), 2014 IEEE International Workshop on*, 2014.

[16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep learning. Book in preparation for MIT Press, 2016.

[17] Ian J. Goodfellow, Oriol Vinyals, and Andrew M. Saxe. Qualitatively characterizing neural network optimization problems. In *ICLR*, 2015.

[18] Slawomir Goryczka, Li Xiong, and Vaidy Sunderam. Secure multiparty aggregation with differential privacy: A comparative study. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, 2013.

[19] Benjamin Graham. Fractional max-pooling. *CoRR*, abs/1412.6071, 2014. URL http://arxiv.org/abs/1412.6071.

[20] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8), November 1997.

[21] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*, 2015.

[22] Yoon Kim, Yacine Jernite, David Sontag, and Alexander M. Rush. Character-aware neural language models. *CoRR*, abs/1508.06615, 2015.

[23] Jakub Konečný, H. Brendan McMahan, Felix X. Yu, Peter Richtarik, Ananda Theertha Suresh, and Dave Bacon. Federated learning: Strategies for improving communication efficiency. In *NIPS Workshop on Private Multi-Party Machine Learning*, 2016.

[24] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

[25] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*. 2012.

[26] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11), 1998.

[27] Chenxin Ma, Virginia Smith, Martin Jaggi, Michael I Jordan, Peter Richtárik, and Martin Takáč. Adding vs. averaging in distributed primal-dual optimization. In *ICML*, 2015.

[28] Ryan McDonald, Keith Hall, and Gideon Mann. Distributed training strategies for the structured perceptron. In *NAACL HLT*, 2010.

[29] Natalia Neverova, Christian Wolf, Griffin Lacey, Lex Fridman, Deepak Chandra, Brandon Barbello, and Graham W. Taylor. Learning human identity from motion patterns. *IEEE Access*, 4:1810–1820, 2016.

[30] Jacob Poushter. Smartphone ownership and internet usage continues to climb in emerging economies. Pew Research Center Report, 2016.

[31] Daniel Povey, Xiaohui Zhang, and Sanjeev Khudanpur. Parallel training of deep neural networks with natural gradient and parameter averaging. In *ICLR Workshop Track*, 2015.

[32] William Shakespeare. The Complete Works of William Shakespeare. Publically available at `https://www.gutenberg.org/ebooks/100`.

[33] Ohad Shamir and Nathan Srebro. Distributed stochastic optimization and learning. In *Communication, Control, and Computing (Allerton)*, 2014.

[34] Ohad Shamir, Nathan Srebro, and Tong Zhang. Communication efficient distributed optimization using an approximate newton-type method. *arXiv preprint arXiv:1312.7853*, 2013.

[35] Reza Shokri and Vitaly Shmatikov. Privacy-preserving deep learning. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, 2015.

[36] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. 15, 2014.

[37] Latanya Sweeney. Simple demographics often identify people uniquely. 2000.

[38] TensorFlow team. Tensorflow convolutional neural networks tutorial, 2016. `http://www.tensorflow.org/tutorials/deep_cnn`.

[39] White House Report. Consumer data privacy in a networked world: A framework for protecting privacy and promoting innovation in the global digital economy. *Journal of Privacy and Confidentiality*, 2013.

[40] Tianbao Yang. Trading computation for communication: Distributed stochastic dual coordinate ascent. In *Advances in Neural Information Processing Systems*, 2013.

[41] Ruiliang Zhang and James Kwok. Asynchronous distributed admm for consensus optimization. In *ICML*. JMLR Workshop and Conference Proceedings, 2014.

[42] Sixin Zhang, Anna E Choromanska, and Yann LeCun. Deep learning with elastic averaging sgd. In *NIPS*. 2015.

[43] Yuchen Zhang and Lin Xiao. Communication-efficient distributed optimization of self-concordant empirical loss. *arXiv preprint arXiv:1501.00263*, 2015.

[44] Yuchen Zhang, Martin J Wainwright, and John C Duchi. Communication-efficient algorithms for statistical optimization. In *NIPS*, 2012.

[45] Yuchen Zhang, John Duchi, Michael I Jordan, and Martin J Wainwright. Information-theoretic lower bounds for distributed statistical estimation with communication constraints. In *Advances in Neural Information Processing Systems*, 2013.

[46] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J. Smola. Parallelized stochastic gradient descent. In *NIPS*. 2010.

A Supplemental Figures and Tables

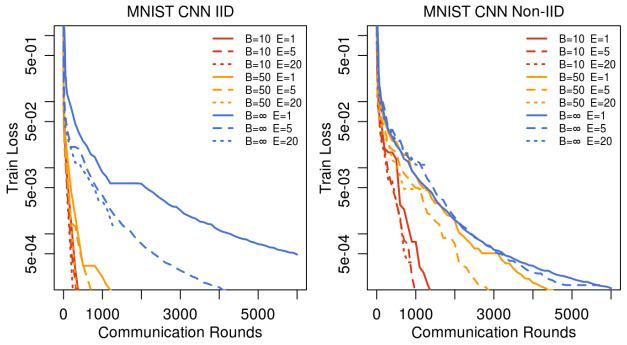


Figure 6: Training set convergence for the MNIST CNN. Note the y-axis is on a log scale, and the x-axis covers more training than Figure 2. These plots fix $C = 0.1$.

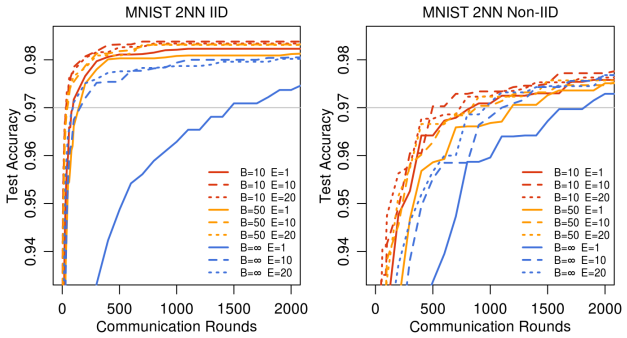


Figure 7: Test set accuracy vs. communication rounds for MNIST 2NN with $C = 0.1$ and optimized η. The left column is the IID dataset, and right is the pathological 2-digits-per-client non-IID data.

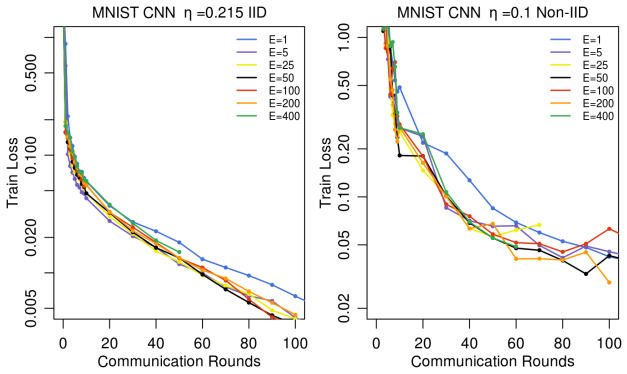


Figure 8: The effect of training for many local epochs (large E) between averaging steps, fixing $B = 10$ and $C = 0.1$. Training loss for the MNIST CNN. Note different learning rates and y-axis scales are used due to the difficulty of our pathological non-IID MNIST dataset.

Table 4: Speedups in the number of communication rounds to reach a target accuracy of 97% for `FedAvg`, versus `FedSGD` (first row) on the MNIST 2NN model.

MNIST 2NN	E	B	u	IID	NON-IID
FEDSGD	1	∞	1	1468	1817
FEDAVG	10	∞	10	156 (9.4×)	1100 (1.7×)
FEDAVG	1	50	12	144 (10.2×)	1183 (1.5×)
FEDAVG	20	∞	20	92 (16.0×)	957 (1.9×)
FEDAVG	1	10	60	92 (16.0×)	831 (2.2×)
FEDAVG	10	50	120	45 (32.6×)	881 (2.1×)
FEDAVG	20	50	240	39 (37.6×)	835 (2.2×)
FEDAVG	10	10	600	34 (43.2×)	497 (3.7×)
FEDAVG	20	10	1200	32 (45.9×)	738 (2.5×)

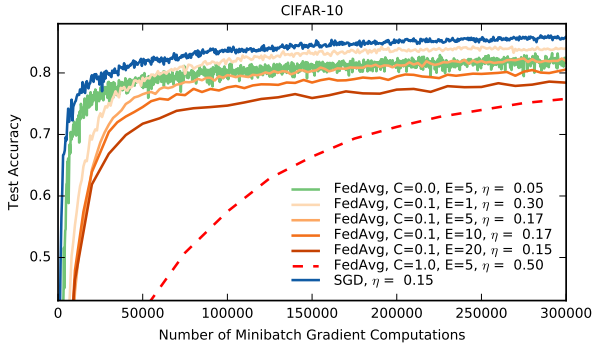


Figure 9: Test accuracy versus number of minibatch gradient computations ($B = 50$). The baseline is standard sequential SGD, as compared to `FedAvg` with different client fractions C (recall $C = 0$ means one client per round), and different numbers of local epochs E.

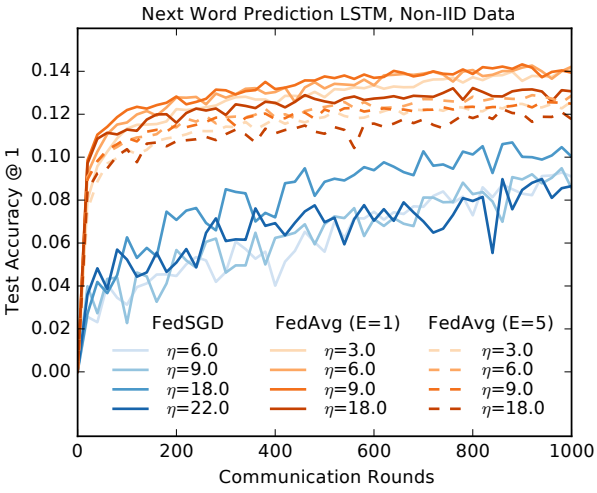


Figure 10: Learning curves for the large-scale language model word LSTM, with evaluation computed every 20 rounds. `FedAvg` actually performs better with fewer local epochs E (1 vs 5), and also has lower variance in accuracy across evaluation rounds compared to `FedSGD`.

PLOS | ONE

RESEARCH ARTICLE

On Pixel-Wise Explanations for Non-Linear Classifier Decisions by Layer-Wise Relevance Propagation

Sebastian Bach[1,2☯]*, Alexander Binder[2,5☯], Grégoire Montavon[2], Frederick Klauschen[3], Klaus-Robert Müller[2,4]*, Wojciech Samek[1,2]*

1 Machine Learning Group, Fraunhofer Heinrich Hertz Institute, Berlin, Germany, **2** Machine Learning Group, Technische Universität Berlin, Berlin, Germany, **3** Charité University Hospital, Berlin, Germany, **4** Department of Brain and Cognitive Engineering, Korea University, Seoul, Korea, **5** ISTD Pillar, Singapore University of Technology and Design (SUTD), Singapore

☯ These authors contributed equally to this work.
* sebastian.bach@hhi.fraunhofer.de (SB), klaus-robert.mueller@tu-berlin.de (KM), wojciech.samek@hhi.fraunhofer.de (WS)

CrossMark
click for updates

OPEN ACCESS

Citation: Bach S, Binder A, Montavon G, Klauschen F, Müller K-R, Samek W (2015) On Pixel-Wise Explanations for Non-Linear Classifier Decisions by Layer-Wise Relevance Propagation. PLoS ONE 10 (7): e0130140. doi:10.1371/journal.pone.0130140

Editor: Oscar Deniz Suarez, Universidad de Castilla-La Mancha, SPAIN

Received: May 19, 2014

Accepted: May 15, 2015

Published: July 10, 2015

Copyright: © 2015 Bach et al. This is an open access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

Data Availability Statement: All relevant data are available from the Harvard Dataverse: http://dx.doi.org/10.7910/DVN/D3GIPK—PASCAL VOC 2009 training and validation data set; http://dx.doi.org/10.7910/DVN/OA9NRS—PASCAL VOC 2009 test; http://dx.doi.org/10.7910/DVN/W3FOUO—the various CC0-images used in figures; http://dx.doi.org/10.7910/DVN/LXIPZF—the polygon data sets for training and LRP, including labels. The same data can still be found at http://doc.ml.tu-berlin.de/layerwise_relevance_propagation/, as previously stated.

Funding: SB was funded by the PROSEC project (http://www.prosec-project.org/index.html, grant no.

Abstract

Understanding and interpreting classification decisions of automated image classification systems is of high value in many applications, as it allows to verify the reasoning of the system and provides additional information to the human expert. Although machine learning methods are solving very successfully a plethora of tasks, they have in most cases the disadvantage of acting as a black box, not providing any information about what made them arrive at a particular decision. This work proposes a general solution to the problem of understanding classification decisions by pixel-wise decomposition of nonlinear classifiers. We introduce a methodology that allows to visualize the contributions of single pixels to predictions for kernel-based classifiers over Bag of Words features and for multilayered neural networks. These pixel contributions can be visualized as heatmaps and are provided to a human expert who can intuitively not only verify the validity of the classification decision, but also focus further analysis on regions of potential interest. We evaluate our method for classifiers trained on PASCAL VOC 2009 images, synthetic image data containing geometric shapes, the MNIST handwritten digits data set and for the pre-trained ImageNet model available as part of the Caffe open source package.

Introduction

Classification of images has become a key ingredient in many computer vision applications, e.g. image search [1], robotics [2], medical imaging [3], object detection in radar images [4] or face detection [5]. Two particularly popular approaches, neural networks [6] and Bag of Words (BoW) models [7], are widely used for these tasks and were among the top submissions in competitions on image classification and ranking such as ImageNet [8], Pascal VOC [9] and ImageCLEF [10]. However, like many methods in machine learning, these models often lack a

16BY1145). The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript. AB and KRM were partially funded by http://www.bmwi.de/EN/root.html (grant no. 01MQ07018). The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript. KRM was partially funded by the BK21 program of the National Research Foundation of Korea in the Brain Korea 21 program (http://www.nrf.re.kr/nrf eng cms/, - no grant number available), and partially funded by the German Ministry for Education and Research as Berlin Big Data Center BBDC, funding mark 01IS14013A. The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript. GM is funded as an employee of the Machine Learning Group of the TU Berlin, http://www.ml.tu-berlin.de/ (research/teaching), http://www.tu-berlin.de/ (no grant number available). The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript. FK is funded by the Human Frontier Science Program (http://www.hfsp.org/, grant no. RGY0077/2011). The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript. WS was funded by the Federal Ministry of Education and Research (BMBF, http://www.bmbf.de/) under the project Adaptive BCI, grant no. 01GQ1115. The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

Competing Interests: The authors of this manuscript have read the journal's policy and have the following competing interests: AB, FK, and KRM have a pending patent application: http://www.google.com/patents/WO2013037983A1?cl = en and http://www.google.com/patents/EP2570970A1?cl = en. This patent deals with pixel-wise visualization of bag of words features. This patent does not deal with artificial neural networks. The authors submitted a second patent application, "RELEVANCE SCORE ASSIGNMENT FOR ARTIFICIAL NEURAL NETWORKS" with the provisional application number PCT/EP2015/056008. The hereby confirm that this does not alter their adherence to PLOS ONE policies on sharing data and materials. All other authors (SB, GM, WS) have declared that no competing interests exist on their behalf.

straightforward interpretability of the classifier predictions. In other words the classifier acts as a *black box* and does not provide detailed information about why it reaches a certain classification decision.

This lack of interpretability is due to the non-linearity of the various mappings that process the raw image pixels to its feature representation and from that to the final classifier function. This is a considerable drawback in classification applications, as it hinders the human experts to carefully verify the classification decision. A simple *yes* or *no* answer is sometimes of limited value in applications, where questions like *where* something occurs or *how* it is structured are more relevant than a binary or real-valued one-dimensional assessment of mere presence or absence of a certain structure.

In this work, we aim to close the gap between classification and interpretability both for multilayered neural networks and Bag of Words (BoW) models over non-linear kernels which are two classes of predictors which enjoy popularity in computer vision. We will consider both types of predictors in a generic sense trying to avoid whenever possible a priori restrictions to specific algorithms or mappings. For the first part, the Bag of Words models will be treated as an aggregation of non-linear mappings over local features in an image which includes a number of popular mapping methods such as Fisher vectors [11], regularized coding [12–14] and soft coding [15], combined with differentiable non-linear kernels and a range of pooling functions including sum- and max-pooling. For the second part, the neural networks (e.g. [6, 16]), we will consider general multilayered network structures with arbitrary continuous neurons and pooling functions based on generalized p-means.

The next Section *Pixel-wise Decomposition as a General Concept* will explain the basic approaches underlying the pixel-wise decomposition of classifiers. In Section *Bag of Words models revisited*, we will give a short recapitulation about Bag of Words features and kernel-based classifiers and summarize related work. *Overview of the decomposition steps* will discuss the decomposition of a kernel-based classifier into sums of scores over small regions of the image, and the projection down to single pixels. Our method then is applied to a number of popular mappings and kernels in Section *Examples for various mappings and kernels*. *Pixel-wise Decomposition for Multilayer Networks* applies both the Taylor-based and layer-wise relevance propagation approaches explained in *Pixel-wise Decomposition as a General Concept* to neural network architectures. The experimental evaluation of our framework will be done in *Experiments* and we conclude the paper with a discussion in *Discussion*.

Pixel-wise Decomposition as a General Concept

The overall idea of pixel-wise decomposition is to understand the contribution of a single pixel of an image x to the prediction $f(x)$ made by a classifier f in an image classification task. We would like to find out, separately for each image x, which pixels contribute to what extent to a positive or negative classification result. Furthermore we want to express this extent quantitatively by a measure. We assume that the classifier has real-valued outputs which are thresholded at zero. In such a setup it is a mapping $f : \mathbb{R}^V \rightarrow \mathbb{R}^1$ such that $f(x) > 0$ denotes presence of the learned structure. Probabilistic outputs can be treated without loss of generality by subtracting 0.5. We are interested to find out the contribution of each input pixel $x_{(d)}$ of an input image x to a particular prediction $f(x)$. The important constraint specific to classification consists in finding the differential contribution relative to the state of maximal uncertainty with respect to classification which is then represented by the set of root points $f(x_0) = 0$. One possible way is to decompose the prediction $f(x)$ as a sum of terms of the separate input dimensions

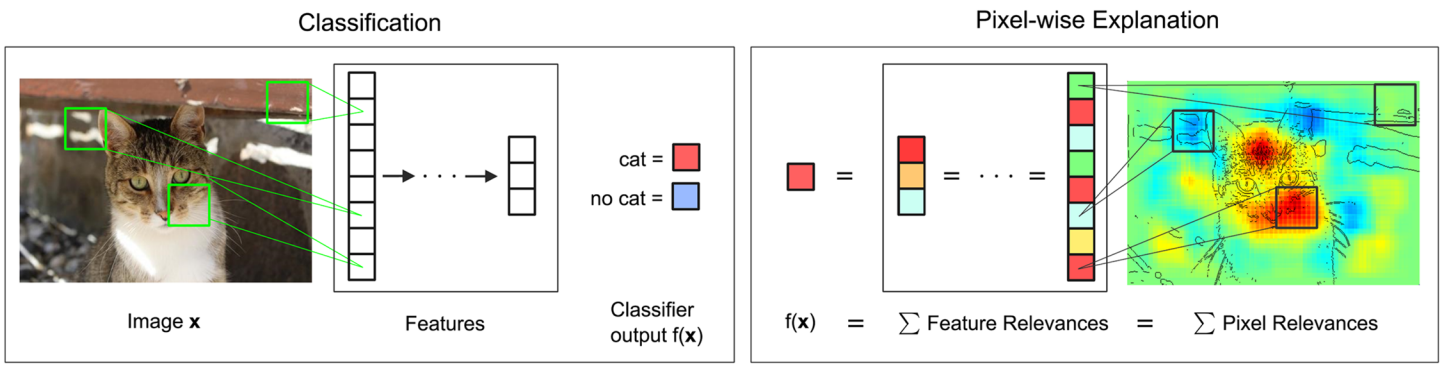


Fig 1. Visualization of the pixel-wise decomposition process. In the classification step the image is converted to a feature vector representation and a classifier is applied to assign the image to a given category, e.g., "cat" or "no cat". Note that the computation of the feature vector usually involves the usage of several intermediate representations. Our method decomposes the classification output $f(x)$ into sums of feature and pixel relevance scores. The final relevances visualize the contributions of single pixels to the prediction. Cat image by pixabay user stinne24.

doi:10.1371/journal.pone.0130140.g001

x_d respectively pixels:

$$f(x) \approx \sum_{d=1}^{V} R_d \tag{1}$$

The qualitative interpretation is that $R_d < 0$ contributes evidence against the presence of a structure which is to be classified while $R_d > 0$ contributes evidence for its presence. In terms of subsequent visualization, which however will not be the scope of this paper, the resulting relevances R_d for each input pixel $x_{(d)}$ can be mapped to a color space and visualized in that way as a conventional heatmap. One basic constraint will be in the following work that the signs of R_d should follow above qualitative interpretation, i.e. positive values should denote positive contributions, negative values negative contributions. Fig 1 depicts the main idea of our method.

In this paper we propose a novel concept we denote as *layer-wise relevance propagation* as a general concept for the purpose of achieving a pixel-wise decomposition as in Eq (1). We also discuss an approach based on *Taylor decomposition* which yields an approximation of layer-wise relevance propagation. We will show that for a wide range of non-linear classification architectures, layer-wise relevance propagation can be done without the need to use an approximation by means of Taylor expansion. The methods we propose in this paper do not involve segmentation. They do not require pixel-wise training as learning setup or pixel-wise labeling for the training phase. The setup used here is image-wise classification, in which during training one label is provided for an image as a whole, however, the contribution of this paper is not about classifier training. The proposed methods are built on top of a pre-trained classifier. They are even applicable to an already pre-trained image classifier as shown in Section *Neural Network for 1000 ILSVRC classes*.

Layer-wise relevance propagation

We will introduce layer-wise relevance propagation as a concept defined by a set of constraints. Any solution satisfying the constraints will be considered to follow the concept of layer-wise relevance propagation. In later sections we will then derive solutions for two particular classifier architectures and evaluate these solutions experimentally for their meaningfulness. Layer-

wise relevance propagation in its general form assumes that the classifier can be decomposed into several layers of computation. Such layers can be parts of the feature extraction from the image or parts of a classification algorithm run on the computed features. As shown later, this is possible for Bag of Words features with non-linear SVMs as well as for neural networks.

The first layer are the inputs, the pixels of the image, the last layer is the real-valued prediction output of the classifier f. The l-th layer is modeled as a vector $z = (z_d^{(l)})_{d=1}^{V(l)}$ with dimensionality $V(l)$. Layer-wise relevance propagation assumes that we have a Relevance score $R_d^{(l+1)}$ for each dimension $z_d^{(l+1)}$ of the vector z at layer $l+1$. The idea is to find a Relevance score $R_d^{(l)}$ for each dimension $z_d^{(l)}$ of the vector z at the next layer l which is closer to the input layer such that the following equation holds.

$$f(x) = \cdots = \sum_{d \in l+1} R_d^{(l+1)} = \sum_{d \in l} R_d^{(l)} = \cdots = \sum_d R_d^{(1)} \tag{2}$$

Iterating Eq (2) from the last layer which is the classifier output $f(x)$ down to the input layer x consisting of image pixels then yields the desired Eq (1). The Relevance for the input layer will serve as the desired sum decomposition in Eq (1). In the following we will derive further constraints beyond Eqs (1) and (2) and motivate them by examples. As we will show now, a decomposition satisfying Eq (2) per se is neither unique, nor it is guaranteed that it yields a meaningful interpretation of the classifier prediction.

We give here a simple counterexample. Suppose we have one layer. The inputs are $x \in \mathbb{R}^V$. We use a linear classifier with some arbitrary and dimension-specific feature space mapping ϕ_d and a bias b

$$f(x) = b + \sum_d \alpha_d \phi_d(x_d) \tag{3}$$

Let us define the relevance for the second layer trivially as $R_1^{(2)} = f(x)$. Then, one possible layer-wise relevance propagation formula would be to define the relevance $R^{(1)}$ for the inputs x as

$$R_d^{(1)} = \begin{cases} f(x) \frac{|\alpha_d \phi_d(x_d)|}{\sum_d |\alpha_d \phi_d(x_d)|} & \text{if } \sum_d |\alpha_d \phi_d(x_d)| \neq 0 \\ \frac{b}{V} & \text{if } \sum_d |\alpha_d \phi_d(x_d)| = 0 \end{cases} \tag{4}$$

This clearly satisfies Eqs (1) and (2), however the Relevances $R^{(1)}(x_d)$ of all input dimensions have the same sign as the prediction $f(x)$. In terms of pixel-wise decomposition interpretation, all inputs point towards the presence of a structure if $f(x) > 0$ and towards the absence of a structure if $f(x) < 0$. This is for many classification problems not a realistic interpretation.

Let us discuss a more meaningful way of defining layer-wise relevance propagation. For this example we define

$$R_d^{(1)} = \frac{b}{V} + \alpha_d \phi_d(x_d) \tag{5}$$

Then, the relevance of a feature dimension x_d depends on the sign of the term in Eq (5). This is for many classification problems a more plausible interpretation. This second example shows that the layer-wise relevance propagation is able to deal with non-linearities such as the feature space mapping ϕ_d to some extent and how an example of layer-wise relevance propagation satisfying Formula (2) may look like in practice. Note that no regularity assumption on the feature

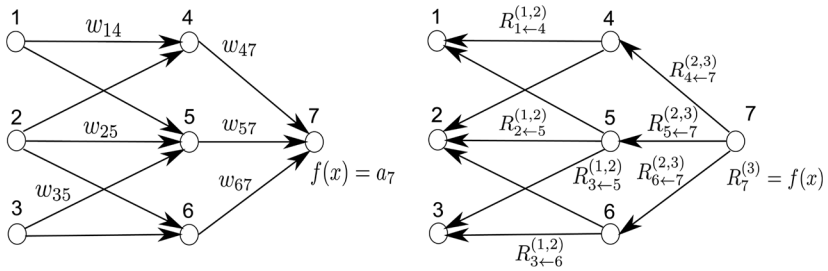


Fig 2. Left: A neural network-shaped classifier during prediction time. w_{ij} are connection weights. a_i is the activation of neuron i. Right: The neural network-shaped classifier during layer-wise relevance computation time. $R_i^{(l)}$ is the relevance of neuron i which is to be computed. In order to facilitate the computation of $R_i^{(l)}$ we introduce messages $R_{i\leftarrow j}^{(l,l+1)}$. $R_{i\leftarrow j}^{(l,l+1)}$ are messages which need to be computed such that the layer-wise relevance in Eq (2) is conserved. The messages are sent from a neuron i to its input neurons j via the connections used for classification, e.g. 2 is an input neuron for neurons 4, 5, 6. Neuron 3 is an input neuron for 5, 6. Neurons 4, 5, 6 are the input for neuron 7.

doi:10.1371/journal.pone.0130140.g002

space mapping ϕ_d is required here at all, it could be even non-continuous, or non-measurable under the Lebesgue measure. The underlying Formula (2) can be interpreted as a conservation law for the relevance R in between layers of the feature processing.

The above example gives furthermore an intuition about what relevance R is, namely, the local contribution to the prediction function $f(x)$. In that sense the relevance of the output layer is the prediction itself $f(x)$. This first example shows what one could expect as a decomposition for the linear case. The linear case is not a novelty, however, it provides a first intuition.

We give a second, more graphic and non-linear, example. The left panel of Fig 2 shows a neural network-shaped classifier with neurons and weights w_{ij} on connections between neurons. Each neuron i has an output a_i from an activation function.

The top layer consists of one output neuron, indexed by 7. For each neuron i we would like to compute a relevance R_i. We initialize the top layer relevance $R_7^{(3)}$ as the function value, thus $R_7^{(3)} = f(x)$. Layer-wise relevance propagation in Eq (2) requires now to hold

$$R_7^{(3)} = R_4^{(2)} + R_5^{(2)} + R_6^{(2)} \tag{6}$$

$$R_4^{(2)} + R_5^{(2)} + R_6^{(2)} = R_1^{(1)} + R_2^{(1)} + R_3^{(1)} \tag{7}$$

We will make two assumptions for this example. Firstly, we express the layer-wise relevance in terms of messages $R_{i\leftarrow j}^{(l,l+1)}$ between neurons i and j which can be sent along each connection. The messages are, however, directed from a neuron towards its input neurons, in contrast to what happens at prediction time, as shown in the right panel of Fig 2. Secondly, we define the relevance of any neuron except neuron 7 as the sum of incoming messages:

$$R_i^{(l)} = \sum_{k:\, i \text{ is input for neuron } k} R_{i\leftarrow k}^{(l,l+1)} \tag{8}$$

For example $R_3^{(1)} = R_{3\leftarrow 5}^{(1,2)} + R_{3\leftarrow 6}^{(1,2)}$. Note that neuron 7 has no incoming messages anyway. Instead its relevance is defined as $R_7^{(3)} = f(x)$. In Eq (8) and the following text the terms *input* and *source* have the meaning of being an input to another neuron in the direction as defined during classification time, not during the time of computation of layer-wise relevance

propagation. For example in Fig 2 neurons 1 and 2 are inputs and source for neuron 4, while neuron 6 is the *sink* for neurons 2 and 3. Given the two assumptions encoded in Eq (8), the layer-wise relevance propagation by Eq (2) can be satisfied by the following sufficient condition:

$$R_7^{(3)} = R_{4\leftarrow 7}^{(2,3)} + R_{5\leftarrow 7}^{(2,3)} + R_{6\leftarrow 7}^{(2,3)} \tag{9}$$

$$R_4^{(2)} = R_{1\leftarrow 4}^{(1,2)} + R_{2\leftarrow 4}^{(1,2)} \tag{10}$$

$$R_5^{(2)} = R_{1\leftarrow 5}^{(1,2)} + R_{2\leftarrow 5}^{(1,2)} + R_{3\leftarrow 5}^{(1,2)} \tag{11}$$

$$R_6^{(2)} = R_{2\leftarrow 6}^{(1,2)} + R_{3\leftarrow 6}^{(1,2)} \tag{12}$$

In general, this condition can be expressed as:

$$R_k^{(l+1)} = \sum_{i:\, i \text{ is input for neuron } k} R_{i\leftarrow k}^{(l,l+1)} \tag{13}$$

The difference between condition (13) and definition (8) is that in the condition (13) the sum runs over the sources at layer l for a fixed neuron k at layer l+1, while in the definition (8) the sum runs over the sinks at layer l+1 for a fixed neuron i at a layer l. When using Eq (8) to define the relevance of a neuron from its messages, then condition (13) is a sufficient condition in order to ensure that Eq (2) holds. Summing over the left hand side in Eq (13) yields

$$\begin{aligned} \sum_k R_k^{(l+1)} &= \sum_k \sum_{i:\, i \text{ is input for neuron } k} R_{i\leftarrow k}^{(l,l+1)} \\ &= \sum_i \sum_{k:\, i \text{ is input for neuron } k} R_{i\leftarrow k}^{(l,l+1)} \\ &\overset{eq.(8)}{=} \sum_i R_i^{(l)} \end{aligned}$$

One can interpret condition (13) by saying that the messages $R_{i\leftarrow k}^{(l,l+1)}$ are used to distribute the relevance $R_k^{(l+1)}$ of a neuron k onto its input neurons at layer l. Our work in the following sections will be based on this notion and the more strict form of relevance conservation as given by definition (8) and condition (13). We set Eqs (8) and (13) as the main constraints defining layer-wise relevance propagation. A solution following this concept is required to define the messages $R_{i\leftarrow k}^{(l,l+1)}$ according to these equations.

Now we can derive an explicit formula for layer-wise relevance propagation for our example by defining the messages $R_{i\leftarrow k}^{(l,l+1)}$. The layer-wise relevance propagation should reflect the messages passed during classification time. We know that during classification time, a neuron i inputs $a_i w_{ik}$ to neuron k, provided that i has a forward connection to k. Thus, we can rewrite the left hand sides of Eqs (9) and (10) so that it matches the structure of the right hand sides of the same equations by the following

$$R_7^{(3)} = R_7^{(3)} \frac{a_4 w_{47}}{\sum_{i=4,5,6} a_i w_{i7}} + R_7^{(3)} \frac{a_5 w_{57}}{\sum_{i=4,5,6} a_i w_{i7}} + R_7^{(3)} \frac{a_6 w_{67}}{\sum_{i=4,5,6} a_i w_{i7}} \tag{14}$$

$$R_4^{(2)} = R_4^{(2)} \frac{a_1 w_{14}}{\sum_{i=1,2} a_i w_{i4}} + R_4^{(2)} \frac{a_2 w_{24}}{\sum_{i=1,2} a_i w_{i4}} \tag{15}$$

The match of the right hand sides of Eqs (9) and (10) against the right hand sides of (14) and (15) can be expressed in general as

$$R_{i\leftarrow k}^{(l,l+1)} = R_k^{(l+1)} \frac{a_i w_{ik}}{\sum_h a_h w_{hk}} \tag{16}$$

While this solution (16) for message terms $R_{i\leftarrow k}^{(l,l+1)}$ still needs to be adapted such that it is usable when the denominator becomes zero, the example given in Eq (16) gives an idea what a message $R_{i\leftarrow k}^{(l,l+1)}$ could be, namely the relevance of a sink neuron $R_k^{(l+1)}$ which has been already computed, weighted proportionally by the input of the neuron i from the preceding layer l. This notion holds in an analogous way when we use different classification architectures and replace the notion of a neuron by a dimension of a feature vector at a given layer.

The Formula (16) has a second property: The sign of the relevance sent by message $R_{i\leftarrow k}^{(l,l+1)}$ becomes inverted if the contribution of a neuron $a_i\, w_{ik}$ has different sign then the sum of the contributions from all input neurons, i.e. if the neuron fires against the overall trend for the top neuron from which it inherits a portion of the relevance. Same as for the example with the linear mapping in Eq (5), an input neuron can inherit positive or negative relevance depending on its input sign. This is a difference to the Eq (4). While this sign switching property can be defined analogously for a range of architectures, we do not add it as a constraint for layer-wise relevance propagation.

One further property is visible here as well. The formula for distribution of relevance is applicable to non-linear and even non-differentiable or non-continuous neuron activations a_k. An algorithm would start with relevances $R^{(l+1)}$ of layer l+1 which have been computed already. Then the messages $R_{i\leftarrow k}^{(l,l+1)}$ would be computed for all elements k from layer l+1 and elements i from the preceding layer l—in a manner such that Eq (13) holds. Then definition (8) would be used to define the relevances $R^{(l)}$ for all elements of layer l.

The relevance conservation property can in principle be supplemented by other constraints that further reduce the set of admissible solutions. For example, one could constrain relevance messages $R_{i\leftarrow k}^{(l,l+1)}$ that result from the redistribution of relevance onto lower-level nodes in a way that is consistent with the contribution of lower-level nodes to the upper layer during the forward pass. If a node i has a larger weighted activation $z_{ik} = a_i w_{ik}$, then, in a qualitative sense, it should also receive a larger fraction of the relevance score $R_k^{(l+1)}$ of the node k. In particular, for all nodes k satisfying R_k, $\Sigma_i\, z_{ik} > 0$, one can define the constraint $0 < z_{ik} < z_{i'k} \Rightarrow R_{i\leftarrow k}^{(l,l+1)} \leq R_{i'\leftarrow k}^{(l,l+1)}$. (The formulas provided in section *Pixel-wise Decomposition for Multilayer Networks* adhere to this ordering constraint.) Nevertheless, nothing is said about the exact relevance values beyond their adherence to the constraint stated above.

To summarize, we have introduced layer-wise relevance propagation in a feed-forward network. In our proposed definition, the total relevance is constrained to be preserved from one layer to another, and the total node relevance must be the equal to the sum of all relevance messages incoming to this node and also equal to the sum of all relevance messages that are outgoing to the same node. It is important to note that the definition is *not* given as an algorithm or a solution of an objective with a distinct minimum. Instead, it is given as a set of constraints that the solution should satisfy. Thus, different algorithms with different resulting solutions may be admissible under these constraints.

Taylor-type decomposition

One alternative approach for achieving a decomposition as in (1) for a general differentiable predictor f is first order Taylor approximation.

$$\begin{aligned} f(x) &\approx f(x_0) + Df(x_0)[x - x_0] \\ &= f(x_0) + \sum_{d=1}^{V} \frac{\partial f}{\partial x_{(d)}}(x_0)(x_{(d)} - x_{0(d)}) \end{aligned} \tag{17}$$

The choice of a Taylor base point x_0 is a free parameter in this setup. As said above, in case of classification we are interested to find out the contribution of each pixel relative to the state of maximal uncertainty of the prediction which is given by the set of points $f(x_0) = 0$, since $f(x) > 0$ denotes presence and $f(x) < 0$ absence of the learned structure. Thus, x_0 should be chosen to be a root of the predictor f. For the sake of precision of the Taylor approximation of the prediction, x_0 should be chosen to be close to x under the Euclidean norm in order to minimize the Taylor residuum according to higher order Taylor approximations. In case of multiple existing roots x_0 with minimal norm, they can be averaged or integrated in order to get an average over all these solutions. The above equation simplifies to

$$f(x) \approx \sum_{d=1}^{V} \frac{\partial f}{\partial x_{(d)}}(x_0)(x_{(d)} - x_{0(d)}) \quad \text{such that } f(x_0) = 0 \tag{18}$$

The pixel-wise decomposition contains a non-linear dependence on the prediction point x beyond the Taylor series, as a close root point x_0 needs to be found. Thus the whole pixel-wise decomposition is not a linear, but a locally linear algorithm, as the root point x_0 depends on the prediction point x.

Several works have been using sensitivity maps [17–19] for visualization of classifier predictions which were based on using partial derivatives *at the prediction point* x. There are two essential differences between sensitivity maps based on derivatives at the prediction point x and the pixel-wise decomposition approach. Firstly, there is no direct relationship between the function value $f(x)$ at the prediction point x and the differential $Df(x)$ at the same point x. Secondly, we are interested in explaining the classifier prediction relative to a certain state given by the set of roots of the prediction function $f(x_0) = 0$. The differential $Df(x)$ at the prediction point does not necessarily point to a root which is close under the Euclidean norm. It points to the nearest local optimum which may still have the same sign as the prediction $f(x)$ and thus be misleading for explaining the difference to the set of root points of the prediction function. Therefore derivatives at the prediction point x are not useful for achieving our aim. Fig 3 illustrates the qualitative difference between local gradients (black arrows) and the dimension-wise decomposition of the prediction (red arrow).

One technical difficulty is to find a root point x_0. For continuous classifiers we may use unlabeled test data in a sampling approach and perform a line search between the prediction point x and a set of candidate points $\{x'\}$ such that their prediction has opposite sign: $f(x)f(x') < 0$. It is clear that the line $l(a) = ax + (1 - a)x'$ must contain a root of f which can be found by interval intersection. Thus each candidate point x' yields one root, and one may select a root point which minimizes the Taylor residuum or use an average over a subset of root points with low Taylor residues. A second possible solution would be, for example, to find the root point x_0 that is the nearest to the data point x, or optimal in some measurable sense. However, while we have presented two types of solutions to finding a root point, in the most general case, the Taylor-type decomposition is best described as a constraint-based approach. In particular, the root point x_0 at which the Taylor decomposition is computed is constrained to satisfy $f(x_0) = 0$ and

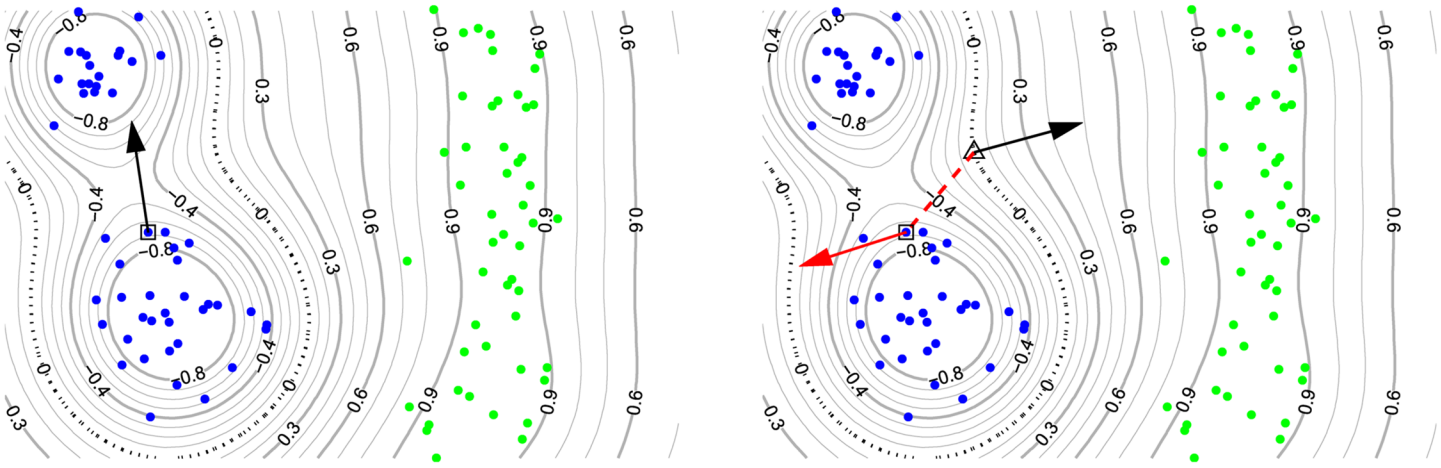


Fig 3. An exemplary real-valued prediction function for classification with the dashed black line being the decision boundary which separates the blue from the green dots. The blue dots are labeled negatively, the green dots are labeled positively. Left: Local gradient of the classification function at the prediction point. Right: Taylor approximation relative to a root point on the decision boundary. This figure depicts the intuition that a gradient at a prediction point x—here indicated by a square—does not necessarily point to a close point on the decision boundary. Instead it may point to a local optimum or to a far away point on the decision boundary. In this example the explanation vector from the local gradient at the prediction point x has a too large contribution in an irrelevant direction. The closest neighbors of the other class can be found at a very different angle. Thus, the local gradient at the prediction point x may not be a good explanation for the contributions of single dimensions to the function value $f(x)$. Local gradients at the prediction point in the left image and the Taylor root point in the right image are indicated by black arrows. The nearest root point x_0 is shown as a triangle on the decision boundary. The red arrow in the right image visualizes the approximation of $f(x)$ by Taylor expansion around the nearest root point x_0. The approximation is given as a vector representing the dimension-wise product between $Df(x_0)$ (the black arrow in the right panel) and $x - x_0$ (the dashed red line in the right panel) which is equivalent to the diagonal of the outer product between $Df(x_0)$ and $x - x_0$.

doi:10.1371/journal.pone.0130140.g003

to lie not too far (e.g. within a fixed radius) from the actual data point x. Using this constraint-based definition, the most desirable properties of the Taylor decomposition are preserved, while the remaining specification is deferred to a later point in time.

Note that Taylor-type decomposition, when applied to one layer or a subset of layers, can be seen as an approximate way of relevance propagation when the function is highly non-linear. This holds in particular when it is applied to the output function f as a function of the preceding layer $f = f(z_{i-1})$, as Eq (18) satisfies approximately the propagation Eq (2) when the relevance of the output layer is initialized as the value of prediction function $f(x)$. Unlike the Taylor approximation, layer-wise relevance propagation does not require to use a second point besides the input point. The formulas in Sections *Pixel-wise Decomposition for Classifiers over Bag of Words Features* and *Pixel-wise Decomposition for Multilayer Networks* will demonstrate that layer-wise relevance propagation can be implemented for a wide range of architectures without the need to approximate by means of Taylor expansion.

Related Work

Several works have been dedicated to the topic of explaining neural networks, kernel-based classifiers in general and classifiers over Bag of Words features in particular.

As for neural networks, [20] is dedicated towards analyzing classifier decisions at neurons which is applicable also to the pixel level. It performs a layer-wise inversion down from output layers towards the input pixels for the architecture of convolutional networks [21]. This work is specific to the architecture of convolutional neural networks. Compared to our approach it is based on a different principle. See [22] which establishes an interpretation of the work in [20]

as an approximation to partial derivatives with respect to pixels in the input image. In a high-level sense, the work in [20] uses the method from their own predecessor work in [23] which solves optimization problems in order to reconstruct the image input, while our approach attempts to reconstruct the classifier decision. In a more technical sense, the difference between [20] and our approach can be seen by comparing how the responses are projected down towards the inputs. [20] uses rectified linear units to project information from the unfolded maps towards the inputs with the aim to ensure the feature maps to be non-negative. In our approach we use the signed activations of the neurons from the layer below for weighting the relevance quantity from the layer above with the aim of conserving the relevance quantity across layers. The sign of the activations of neurons of the layer below, for which the relevance is to be computed, is used in our work for encoding assumptions about the structure of the neural network. With respect to neural networks, our work is applicable to a wide range of architectures. A different type of analysis for understanding neural networks deals with creating counter-intuitive examples, see for example the work in [24, 25]. For a treatment on how to discriminate structures which influence predictions from correlations with hidden variables such that the hidden variables have an impact on the predictor, see [26].

Another approach which lies between partial derivatives at the input point x and a full Taylor series around a different point x_0 is presented in [22]. This work uses a different point x_0 than the input point x for computing the derivative and a remainder bias which both are not specified further but avoids for an unspecified reason to use the full linear weighting term $x - x_0$ of a Taylor series. Besides deviating in the usage of a Taylor series, the work in [22] does not make use of the layer-wise propagation strategy for neural networks presented in this paper, which unlike a Taylor series does not rely on a local approximation. Explanation of neural network behavior on the level of single neurons is done in [27] and [28]. These works try to find inputs which maximize the activation of neurons by means of optimization problems which can be solved by gradient ascent. Our approach aims at explaining the decision for a given input rather than finding optimal stimuli for a particular neuron. Note that the approach presented in our paper is different from plotting the activations of neurons during a forward-pass of the input image. The latter is independent from the neural network properties in higher layers whereas in our approach we feed the classification score into the top neurons and use quantities computed by using properties of higher layers to obtain a representation at lower layers. Quantifying the importance of input variables using a neural network model has also been studied in specific areas such as ecological modeling, where [29, 30] surveyed a large ensemble of possible analyses, including, computing partial derivatives, perturbation analysis, weights analysis, and studying the effect of including and removing variables at training time. A different avenue to understanding decisions in neural network is to fit a more interpretable model (e.g. decision tree) to the function learned by the neural network [31], and extract the rules learned by this new model.

With respect to sensitivity of kernel-based classifiers to input dimensions, [32] yields sensitivity maps, which are invariant to the sign of the predictions. [17–19] consider explanation vectors based on local gradients which are sign-sensitive. Both approaches use partial derivatives at the point which is to be predicted. The first works to our knowledge to visualize BoW models are [33] and [34]. The former work finds regions with a large influence for classification for the special case of max pooling, sparse coding and a linear classifier. The latter work obtains an exact decomposition into local feature scores for the case of a histogram intersection kernel and zero-one coding of local features onto visual words.

We differ from the above works on kernel-based classifiers over Bag of Words features in the following sense: Our methodology is applicable to *arbitrary* Bag of Words models including various feature codings such as Fisher vectors and regularized codings and to a broader class of

kernels, namely all differentiable or so-called sum decomposable kernels. When relying on Taylor decomposition, our work relies on a Taylor series around a root close to the prediction point rather than partial derivatives at the prediction point itself. The rationale for this choice was justified in the preceding Section *Taylor-type decomposition*. Finally, in contrast to the preceding publications, our work introduces layer-wise relevance propagation for neural networks and Bag of Words features.

Pixel-wise Decomposition for Classifiers over Bag of Words Features

Despite recent advances in neural networks, Bag of Words models are still popular for image classification tasks. They have excelled in past competitions on visual concept recognition and ranking such as Pascal VOC [35, 36] and ImageCLEF PhotoAnnotation [37]. In our experience Bag of Words Models perform well for tasks with small sample sizes, whereas neural networks are at risk to overfit due to their richer parameter structure.

Bag of Words models revisited

We will consider here Bag of Words features as an aggregation of non-linear mappings of local features. All Bag of Words models, no matter whether based on hierarchical clustering [38], soft codebook mapping [15, 39–41], regularized local codings [12–14], or Fisher Vectors [11], share a multi-stage procedure in common.

In the first stage local features are computed across small regions in the image. A local feature such as SIFT [42, 43] is in the abstract sense a vector computed from a region of the image, for example capturing information of interest such as shape characteristics or properties of color or texture on a local scale. In a second stage which is performed once during training, representatives in the space of local features are computed, no matter whether they are cluster centroids obtained from k-means clustering, regions of the space as for clustering trees [38], or centers of distributions as for Fisher vectors [44]. The set of representatives, in the following referred to as *visual words*, serves as a vocabulary in the context of which images can be described as vectors. In the third stage, statistics of the local features are computed relative to those visual words. These statistics are aggregated from all local features l within an image in order to yield a BoW representation x, usually done by sum- or max-pooling.

The computation of statistics can be modeled by a mapping function accepting local feature vectors l as input, which are then projected into the Bag of Words feature space. Let m be such a mapping function and let $m_{(d)}$ denote the mapping onto the d-th dimension of the BoW space. We assume the very generic p-means mapping scheme for local features l as given in Eq (19).

$$x_{(d)} = \left(M^{-1} \sum_{j=1}^{M} (m_{(d)}(l_j))^p \right)^{\frac{1}{p}} \tag{19}$$

This contains sum- and max-pooling as the special cases $p = 1$ and the limit $p = \infty$.

Finally, a classifier is applied on top of these features. Our method supports the general class of classifiers based on kernel methods. For brevity we use here an SVM prediction function which results in a prediction function over BoW features x_i, training data labels y_i, kernel

Table 1. Notation Conventions Used in This Section.

$f(\cdot)$	the classifier's prediction function
x,y	BoW representation and class label
x_0,z	root point of Taylor Expansion, root point candidates
α,b	learned model parameters
$k(x_i,x_{i'})$	kernel function
d,V	counter and number of BoW-dimensions
$R_d^{(3)}$	(approximate) contribution of BoW-dimension d
$R_l^{(2)}$	local feature relevance
$R_q^{(1)}$	pixel-wise decompositions per pixel q
$m(l)$	mapping function between local features l and BoW
l,l'	local feature descriptors
$Z(x)$	the set of unmapped dimensions of a BoW data point x
area(l)	the set of pixel coordinates covered by l

doi:10.1371/journal.pone.0130140.t001

functions $k(\cdot,\cdot)$, and SVM model parameters b and α_i.

$$f(x) = b + \sum_{i=1}^{S} \alpha_i y_i k(x_i, x) \tag{20}$$

This assumption can be extended without loss of generality to approaches using multiple kernel functions such as multiple kernel learning [45–49], structural prediction approaches with tensor product structure between features and labels as in taxonomy-based classifiers [50–52] or boosting-like formulations as in [53]

$$f(x) = b + \sum_{i=1}^{S} \sum_{u=1}^{K} \alpha_{i,u} k_u(x_{i(u)}, x_{(u)}). \tag{21}$$

An overview over notations used for the Bag of Words features is given in Table 1.

Overview of the decomposition steps

The main contribution of this part is the formulation of a generic framework for retracing the origins of a decision made by the learned kernel-based classifier function for a BoW feature. This is achieved, in a broad sense as visualized in Fig 4, by following the construction of a BoW representation x of an image and the evaluation thereof by a classifier function in reverse direction. In this section we will derive a decomposition of a kernel-based classifier prediction into contributions of individual local features and finally single pixels. The proposed approach consists of three consecutive steps.

In the first step we will use, depending on the type of kernel, either the Taylor-type decomposition strategy or the layer-wise relevance propagation strategy. In the first step relevance scores $R_d^{(3)}$ for the third layer of the BoW feature extraction process are obtained, describing the influence of all BoW feature dimensions d by deconstructing the classifier prediction function $f(x)$ such that $\sum_d R_d^{(3)} \approx f(x)$. In other words, we gain a decomposition into contributions $R_d^{(3)}$ describing $f(x)$ as a sum of individual predictions for all dimensions of x.

In the second step we will apply the layer-wise relevance propagation strategy in order to obtain relevance scores $R_l^{(2)}$ for the local features l from the relevance scores $R_d^{(3)}$. Layer-wise relevance propagation ensures that $\sum_l R_l^{(2)} = \sum_d R_d^{(3)} \approx f(x)$ holds.

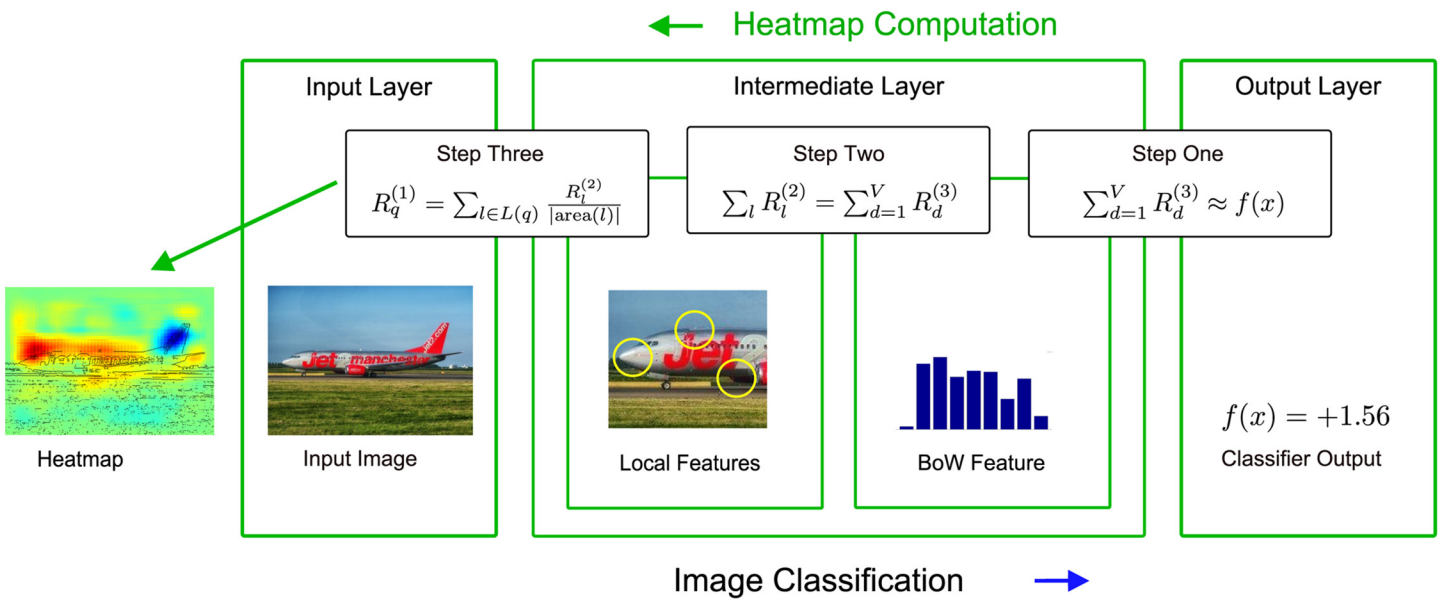


Fig 4. Local and global predictions for input images are obtained by following a series of steps through the classification- and pixel-wise decomposition pipelines. Each step taken towards the final pixel-wise decomposition has a complementing analogue within the Bag of Words classification pipeline. The calculations used during the pixel-wise decomposition process make use of information extracted by those corresponding analogues. Airplane image in the graphic by Pixabay user tpsdave.

doi:10.1371/journal.pone.0130140.g004

The third step describes the computation of pixel-wise scores $R_q^{(1)}$ from local feature relevance scores $R_l^{(2)}$, which are then visualized as heatmaps by color-coding.

Step one: relevance scores $R_d^{(3)}$ for the third layer of the BoW feature extraction process. The third layer is the BoW feature itself. In the first step we would like to achieve a decomposition of the classifier prediction $f(x)$ into relevance scores $R_d^{(3)}$ for BoW feature dimension d.

$$f(x) \approx \sum_{d=1}^{V} R_d^{(3)} \tag{22}$$

The work of [34] has performed this step for the special case of one single histogram intersection kernel. Such a decomposition can be generalized naturally and performed without error for all kernel functions which are sum-decomposable along input dimensions. We define a kernel function k to be sum-decomposable if there exists kernel functions $k^{(d)}$ acting on single input feature dimensions such that

$$k(x_i, x_{i'}) = \sum_{(d)} k^{(d)}(x_{i(d)}, x_{i'(d)}) \tag{23}$$

In this case, as with the linear and histogram intersection kernel, we can achieve Eq (22) with equality in the following definition by applying layer-wise relevance propagation which results in Eq (24).

Def. 1 *Relevance scores for sum decomposable kernels*

$$R_d^{(3)} = \frac{b}{V} + \sum_{i=1}^{S} \alpha_i y_i k^{(d)}(x_{i(d)}, x_{(d)}) \tag{24}$$

For the case of a general differentiable kernel we apply the Taylor-type decomposition strategy in order to linearly approximate the dimensional contributions $R_d^{(3)}$. The approximated dimensional contributions can be expressed as in Eq (25).

Def. 2 *Relevance scores for differentiable kernels*

$$R_d^{(3)} \quad := \quad (x - x_0)_{(d)} \sum_{i=1}^{S} \alpha_i y_i \frac{\partial k(x_i, \cdot)}{\partial x_{(d)}}(x_0) \tag{25}$$

Step two: relevance scores $R_l^{(2)}$ for the second layer of the BoW feature extraction process. The second layer are the local features extracted from many regions of the image. In the second step we would like to achieve a decomposition of the classifier prediction $f(x)$ into relevance scores $R_l^{(2)}$ for the local features l based on the relevances $R_d^{(3)}$ from the third layer.

For the sake of clarity, we do for now start with the case of sum-pooled BoW aggregation, to later extend to a more general formulation for p-means pooling from this point on.

As introduced in context of Eq (19) $m_{(d)}$ denotes the mapping projecting to dimension d of the BoW space. We define the set of input dimensions $Z(x)$ which are effectively not reached by the mappings of local features of an image. Applying the layer-wise propagation allows to define the local feature relevance score $R_l^{(2)}$ as given in Eq (27).

Def. 3 *Local feature scores for sum pooling*

$$Z(x) = \left\{ d \mid \sum_l m_{(d)}(l) = 0 \right\} \tag{26}$$

$$R_l^{(2)} \quad := \quad \sum_{d \notin Z(x)} R_d^{(3)} \frac{m_{(d)}(l)}{\sum_{l'} m_{(d)}(l')} + \sum_{d \in Z(x)} R_d^{(3)} \frac{1}{|\{l'\}|} \tag{27}$$

The coarse structure of definition (27) can be explained as taking the relevances from the layer above and weighting them with the outputs from the layer below. The summations over the mappings $m_{(d)}(l)$ over the local features $\{l\}$ in Eq (27) achieve a weighting of the relevance from the third layer proportional to the ratios of the mappings. The second part describes an equal distribution of those relevance scores $R_d^{(3)}$ which correspond to dimensions of the Bag of Words feature x which have a value of zero and yet may contribute to the classifier prediction. To see this consider computing the χ^2-kernel value to a support vector which has a non-zero value in the same feature dimension. This is necessary to ensure that the propagation Eq (2) holds.

Summing the local feature relevance scores $R_l^{(2)}$ from Eq (27) yields the Taylor approximation $R_d^{(3)}$ of the prediction score $f(x)$ in Eqs (24) or (25). This property is the key to our approach. We obtain exact summation to the prediction $f(x)$ in the case of sum decomposable kernels and usage of Eq (24) in this special case.

$$\sum_l R_l^{(2)} = \sum_{d=1}^{V} R_d^{(3)} \approx f(x) \tag{28}$$

We would like to point out that this property holds also in the case when mappings $m_{(d)}$ can

become negative as a consequence of the definition used in (26) and (27), as can be seen from the summation given in the appendix. For that reason our approach is also applicable to Fisher vectors [11] and regularized coding approaches [12–14]. Furthermore note that definition (27) has no explicit dependence on the way how the local features are pooled in Eq (19) and this might be inappropriate weighting for max-pooling or general p-means pooling.

We can extend this definition to reflect the usage of p-means pooling

$$M_p(x_1, \ldots, x_n) \quad = \left(\frac{1}{n}\sum_{i=1}^{n} x_i^p\right)^{1/p} \tag{29}$$

which may yield different results in prediction and local decomposition than sum pooling. The extension is well-defined for non-negative mappings $m_{(d)} \geq 0$ and any value of p and for arbitrary mappings when combined with a value of p from the natural numbers.

Def. 4 ***Local feature scores for p-means pooling***

$$Z^{(p)}(x) = \left\{ d \mid \sum_l m_{(d)}^p(l) = 0 \right\} \tag{30}$$

$$R_l^{(2)} \quad := \sum_{d \notin Z^{(p)}(x)} R_d^{(3)} \frac{m_{(d)}^p(l)}{\sum_{l'} m_{(d)}^p(l')} + \sum_{d \in Z^{(p)}(x)} R_d^{(3)} \frac{1}{|\{l'\}|} \tag{31}$$

The first quotient in Eq (31) converges to an indicator function for the maximal mapping element in the limit $p \to \infty$ which is consistent to max-pooling

$$\frac{m_{(d)}^p(l)}{\sum_{l'} m_{(d)}^p(l')} \to \mathbb{I}_{\{\text{argmax}_{l'} m_{(d)}(l')\}}(l) \tag{32}$$

Step three: relevance scores $R_q^{(1)}$ for the first layer of the BoW feature extraction process. The first layer are the pixels of the image. In order to calculate scores for each pixel we make use of information regarding local feature geometry and location known from the local feature extraction phase at the beginning of the image classification pipeline. The pixel score $R_q^{(1)}$ of an image coordinate q is calculated as a sum of local feature scores of all local features l covering q, weighted by the number of pixels covered by each local feature l. In terms of layer-wise relevance propagation a local feature is a computation unit which has as much inputs as the number of pixels it is covering. Without assumption of any further structure we distribute the relevance of the local feature equally to all its covered pixels. Layer-wise propagation yields Eq (34).

$$L(q) = \{l \mid q \in \text{area}(l)\} \tag{33}$$

$$R_q^{(1)} = \sum_{l \in L(q)} \frac{R_l^{(2)}}{|\text{area}(l)|} \tag{34}$$

The operator area(l) used Eqs (33) and (34) returns the set of pixel coordinates covered by a local feature l. Projecting the obtained scores $R_q^{(1)}$ to their respective image coordinates yields the pixel-wise decomposition representation h of the evaluated image.

For visualization in the sense of color coding, the pixel-wise decomposition $R^{(1)}$ is then normalized as

$$R'^{(1)}_q = \frac{R^{(1)}_q}{\max_{q'}(|R^{(1)}_{q'}|)} \tag{35}$$

ensuring that $\forall q : R'^{(1)}_q \in [-1, 1]$. The normalized pixel-wise decomposition is then color coded by mapping the pixel scores to a color space of choice. In the case of individually classified sub-regions originating from the same image, a global pixel-wise decomposition can be constructed by averaging local pixel-wise decompositions scores.

Note that by choosing above normalization scheme, the assumption is made that at least one class is represented within the image. In case the assumption holds this might lead to prominent local predictions of even weakly projected features which we found suitable for the purpose of detecting class evidence. If this assumption does not hold, then images may display score artifacts dominated by the set of pixels covered by a small subset of local features which would otherwise be considered input noise. A solution for this problem is global normalization which uses a maximum over pixels over a set of images instead of one image. We found that a global normalization scheme can be more appropriate for visualizing the actual decision process of the classifier, as it preserves the relative order of magnitude of local feature scores in between pixel-wise decomposition tiles. Algorithm 1 gives an overview how to compute the pixel-wise decomposition for classifiers based on Bag of Words features and support vector machines.

Algorithm 1 Pixel-wise decomposition for BoW features with SVM classifiers
Inputs:
Image I
Local features L
BoW representation x (and Taylor root point x_0)
model and mapping parameters
for $d = 1$ **to** V **do**
    $R^{(3)}_d$ as in Eqs (24) or (25)
end for
for all $l \in L$ **do**
    $R^{(2)}_l$ as in Eqs (27) or (31)
end for
for all pixels $q \in I$ **do**
    $R^{(1)}_q$ as in Eq (34)
end for
Output: $\forall q : R^{(1)}_q$

Examples for various mappings and kernels

In order to illustrate the generality of this framework we give some examples for various methods of mapping local features and kernels.

Example case: Soft codebook mapping. A soft codebook mapping like in [15, 40] fits trivially into the framework

$$m_{(d)}(l) = \frac{\exp(-\beta d(l, b_d))}{\sum_{(d)} \exp(-\beta d(l, b_d))} \tag{36}$$

Example case: Regularized coding. Considering formulations as in [12, 14]

$$\text{argmin}_{B,C} \sum_j \| l_j - Bc_j \|^2 + Q(c_j) \tag{37}$$

where B is the codebook and Q denotes a regularizer on the codebook coefficients such as the ℓ_1-norm $Q(c) = \sum_j |c(j)|$, we arrive at

$$m(l_j) := c_j \tag{38}$$

where the d-th dimension of the so defined mapping $m_{(d)}$ corresponds to the d-th basis element in B.

Example case: Fisher Vectors. The Fisher vector [44] can be replicated using the following mapping $m_{(d)}$ given below. It is expected to be used with a linear kernel, thus we can use the exact formula from Eq (24) for defining the BoW dimension relevance scores $R_d^{(3)}$. We stick to the formulation of Fisher vectors as given in [11]. The difference to other mappings is that a BoW feature dimension corresponds to a derivative of a Gaussian mixture center with respect to some parameter and not to a visual word.

Let $g_k(l)$ be the D-dimensional Gaussian mixture component with diagonal covariance σ_k and mean μ_k. We define the softmax of the GMM mixture parameters α_k as

$$w_k = \frac{\exp(\alpha_k)}{\sum_{j=1}^{K} \exp(\alpha_j)} \tag{39}$$

Let the soft assignment of local feature l to Gaussian k be given as

$$\gamma[l](k) = \frac{w_k u_k(l)}{\sum_{j=1}^{K} w_j u_j(l)} \tag{40}$$

Let k denote the index of a Gaussian mixture component, then the mapping contains three cases depending on whether we consider the derivative for the mixture parameter α_k stored in dimension $d = (2D + 1)(k - 1) + 1$, derivatives for the Gaussian mean parameter μ_k stored in dimensions $d = (2D + 1)(k - 1) + 1 + r, r \in [1,D]$, or finally, derivatives for the Gaussian variance parameter σ_k stored in dimensions $d = (2D + 1)(k - 1) + 1 + D + r, r \in [1,D]$

We assume one kernel without loss of generality, $V_1 = (2D + 1)K$ where K is the number of Gaussian mixture components and D is the local feature dimension.

$$m_{(d)}(l) = \begin{cases} \frac{1}{\sqrt{w_k}} (\gamma[l](k) - w_k) & \text{if } d = (2D+1)(k-1)+1 \\ \frac{1}{\sqrt{w_k}} \gamma[l](k) \frac{l_r - \mu_k}{\sigma_k} & \text{if } d = (2D+1)(k-1)+1+r, r \in [1,D] \\ \frac{1}{\sqrt{w_k}} \gamma[l](k) \frac{1}{\sqrt{2}} \left(\frac{(l_r - \mu_k)^2}{\sigma_k^2} - 1 \right) & \text{if } d = (2D+1)(k-1)+1+D+r, r \in [1,D] \end{cases} \tag{41}$$

This definition can be extended in a straightforward manner to use power normalization and ℓ_2 normalization from [11]. The difference to the soft mapping case from Section *Example case: Soft codebook mapping* is that a the d-th dimension of the mapping $m_{(d)}$ corresponds not to a visual word but to a derivative with respect to a Gaussian mixture component.

Example case: Histogram intersection kernel. The histogram intersection kernel applies to the exact decomposition formula in Eq (24). It is defined as

$$k_{\mathrm{HI}}(x_i, x_{i'}) = \sum_d \min\left(x_{i(d)}, x_{i'(d)}\right) \tag{42}$$

Plugging the kernel into (24) yields

$$R_d^{(3)} = \frac{b}{V} + \sum_{i=1}^{S} \alpha_i y_i \min\left(x_{i(d)}, x_{(d)}\right) \tag{43}$$

Example case: χ^2 kernel

The χ^2 kernel function has been applied with good performance in various image classification tasks [41, 54, 55]. The χ^2 kernel is defined as

$$k_{\chi^2}(x_i, x_{i'}) = \exp\left(-\sigma \sum_d \frac{\left(x_{i(d)} - x_{i'(d)}\right)^2}{\left(x_{i(d)} + x_{i'(d)}\right)}\right) \tag{44}$$

To avoid divisions by zero in case of $x_{i(d)} = x_{i'(d)} = 0$ we adhere to the convention that $\frac{0}{0} = 0$ for the kernel function itself, as well as its derivative

$$\frac{\partial k_{\chi^2}(x_i, x_{i'})}{\partial x_{i'(d)}} = k_{\chi^2}(x_i, x_{i'})\sigma\left(\frac{4\left(x_{i(d)}\right)^2}{\left(x_{i(d)} + x_{i'(d)}\right)^2} - 1\right) \tag{45}$$

When plugged into Eq (25), the dimensional contributions of the χ^2 kernel are then obtained as

$$R_d^{(3)} := (x - x_0)_{(d)} \cdot \sum_{i=1}^{S} \alpha_i y_i k_{\chi^2}(x_i, x_0)\sigma\left(\frac{4\left(x_{i(d)}\right)^2}{\left(x_{i(d)} + x_{0(d)}\right)^2} - 1\right) \tag{46}$$

Example case: Gaussian-RBF kernel. The Gaussian-RBF kernel function is widely used in different communities and is one of the most prominent, if not the most prominent non-linear kernel function. The kernel function is defined as

$$k_{\mathrm{Gauss}}(x_i, x_{i'}) = \exp\left(-\frac{\| x_i - x_{i'} \|_2^2}{\sigma}\right) \tag{47}$$

Deriving the kernel function w.r.t $x_{i'(d)}$ and plugging it into Eq (25) yields

$$\frac{\partial k_{\mathrm{Gauss}}(x_i, x_{i'})}{\partial x_{i'(d)}} = k_{\mathrm{Gauss}}(x_i, x_{i'})\left(\frac{2x_{i(d)} - 2x_{i'(d)}}{\sigma}\right) \tag{48}$$

and consequently

$$R_d^{(3)} := (x - x_0)_{(d)} \cdot \sum_{i=1}^{S} \alpha_i y_i k_{\mathrm{Gauss}}(x_i, x_0)\left(\frac{2x_{i(d)} - 2x_{0(d)}}{\sigma}\right) \tag{49}$$

Pixel-wise Decomposition for Multilayer Networks

Multilayer networks are commonly built as a set of interconnected neurons organized in a layer-wise manner. They define a mathematical function when combined to each other, that maps the first layer neurons (input) to the last layer neurons (output). We denote each neuron by x_i where i is an index for the neuron. By convention, we associate different indices for each layer of the network. We denote by "Σ_i" the summation over all neurons of a given layer, and by "Σ_j" the summation over all neurons of another layer. We denote by $x_{(d)}$ the neurons corresponding to the pixel activations (i.e. with which we would like to obtain a decomposition of the classification decision). A common mapping from one layer to the next one consists of a linear projection followed by a non-linear function:

$$z_{ij} = x_i w_{ij}, \tag{50}$$

$$z_j = \sum_i z_{ij} + b_j, \tag{51}$$

$$x_j = g(z_j), \tag{52}$$

where w_{ij} is a weight connecting the neuron x_i to neuron x_j, b_j is a bias term, and g is a non-linear activation function. Multilayer networks stack several of these layers, each of them, composed of a large number of neurons. Common non-linear functions are the hyperbolic tangent $g(t) = \tanh(t)$ or the rectification function $g(t) = \max(0,t)$. This formulation of neural network is general enough to encompass a wide range of architectures such as the simple multilayer perceptron [56] or convolutional neural networks [57], where convolution and sum-pooling are linear operations.

Taylor-type decomposition

Denoting by $f{:}\mathbb{R}^M \mapsto \mathbb{R}^N$ the vector-valued multivariate function implementing the mapping between input and output of the network, a first possible explanation of the classification decision $x \mapsto f(x)$ can be obtained by Taylor expansion at a near root point x_0 of the decision function f:

$$R_d^{(1)} = (x - x_0)_{(d)} \cdot \frac{\partial f}{\partial x_{(d)}}(x_0) \tag{53}$$

The derivative $\partial f(x)/\partial x_{(d)}$ required for pixel-wise decomposition can be computed efficiently by reusing the network topology using the backpropagation algorithm [56]. In particular, having backpropagated the derivatives up to a certain layer j, we can compute the derivative of the previous layer i using the chain rule:

$$\frac{\partial f}{\partial x_i} = \sum_j \frac{\partial f}{\partial x_j} \cdot \frac{\partial x_j}{\partial x_i} = \sum_j \frac{\partial f}{\partial x_j} \cdot w_{ij} \cdot g'(z_j). \tag{54}$$

A requirement of the Taylor-based decomposition is to find roots x_0 (i.e. points on the classification boundary) that support a local explanation of the classification decision for x. These roots can be found by local search in the neighborhood of x. However, as noted in [24], this can lead to points of the input space that are perceptually equivalent to the original sample x and whose choice as a root would produce non-informative pixel-wise decompositions.

Alternatively, root points can be found by line search on the segment defined by x and its closest neighbor of a different class. This solution is problematic when the data manifold is sparsely populated, as it is the case for natural images. In such case, it is likely that following a

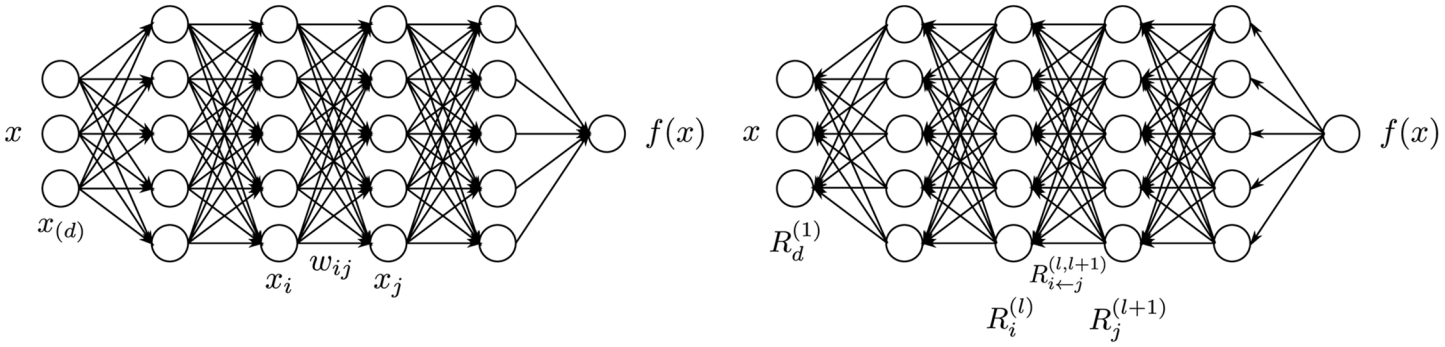


Fig 5. Multilayer neural network annotated with the different variables and indices describing neurons and weight connections. Left: forward pass. Right: backward pass.

doi:10.1371/journal.pone.0130140.g005

straight line between x and its nearest neighbor will strongly depart from the data manifold and produce roots x_0 with similarly poor pixel-wise decompositions.

Layer-wise relevance backpropagation

As an alternative to Taylor-type decomposition, it is possible to compute relevances at each layer in a backward pass, that is, express relevances $R_i^{(l)}$ as a function of upper-layer relevances $R_j^{(l+1)}$, and backpropagating relevances until we reach the input (pixels). Fig 5 depicts a graphic example.

The method works as follows: Knowing the relevance of a certain neuron $R_j^{(l+1)}$ for the classification decision $f(x)$, one would like to obtain a decomposition of such relevance in terms of messages sent to neurons of the previous layers. We call these messages $R_{i \leftarrow j}$. In particular, as expressed by Eqs (8) and (13), the conservation property

$$\sum_i R_{i\leftarrow j}^{(l,l+1)} = R_j^{(l+1)} \tag{55}$$

must hold. In the case of a linear network $f(x) = \sum_i z_{ij}$ where the relevance $R_j = f(x)$, such decomposition is immediately given by $R_{i \leftarrow j} = z_{ij}$. However, in the general case, the neuron activation x_j is a non-linear function of z_j. Nevertheless, for the hyperbolic tangent and the rectifying function—two simple monotonically increasing functions satisfying $g(0) = 0$, —the pre-activations z_{ij} still provide a sensible way to measure the relative contribution of each neuron x_i to R_j. A first possible choice of relevance decomposition is based on the ratio of local and global pre-activations and is given by:

$$R_{i\leftarrow j}^{(l,l+1)} = \frac{z_{ij}}{z_j} \cdot R_j^{(l+1)} \tag{56}$$

These relevances $R_{i \leftarrow j}$ are easily shown to approximate the conservation properties of Eq (2), in particular:

$$\sum_i R_{i\leftarrow j}^{(l,l+1)} = R_j^{(l+1)} \cdot \left(1 - \frac{b_j}{z_j}\right) \tag{57}$$

where the multiplier accounts for the relevance that is absorbed (or injected) by the bias term. If necessary, the residual bias relevance can be redistributed onto each neuron x_i.

A drawback of the propagation rule of Eq (56) is that for small values z_j, relevances $R_{i \leftarrow j}$ can take unbounded values. Unboundedness can be overcome by introducing a predefined stabilizer $\varepsilon \geq 0$:

$$R_{i\leftarrow j}^{(l,l+1)} = \begin{cases} \frac{z_{ij}}{z_j+\varepsilon} \cdot R_j^{(l+1)} & z_j \geq 0 \\ \frac{z_{ij}}{z_j-\varepsilon} \cdot R_j^{(l+1)} & z_j < 0 \end{cases} \tag{58}$$

The conservation law then becomes

$$\sum_i R_{i\leftarrow j}^{(l,l+1)} = \begin{cases} R_j^{(l+1)} \cdot \left(1 - \frac{b_j+\varepsilon}{z_j+\varepsilon}\right) & z_j \geq 0 \\ R_j^{(l+1)} \cdot \left(1 - \frac{b_j-\varepsilon}{z_j-\varepsilon}\right) & z_j < 0 \end{cases} \tag{59}$$

where we can observe that some further relevance is absorbed by the stabilizer. In particular, relevance is fully absorbed if the stabilizer ε becomes very large.

An alternative stabilizing method that does not leak relevance consists of treating negative and positive pre-activations separately. Let $z_j^+ = \sum_i z_{ij}^+ + b_j^+$ and $z_j^- = \sum_i z_{ij}^- + b_j^-$ where " $-$ " and "+" denote the negative and positive part of z_{ij} and b_j. Relevance propagation is now defined as

$$R_{i\leftarrow j}^{(l,l+1)} = R_j^{(l+1)} \cdot \left(\alpha \cdot \frac{z_{ij}^+}{z_j^+} + \beta \cdot \frac{z_{ij}^-}{z_j^-}\right) \tag{60}$$

where $\alpha + \beta = 1$. For example, for $\alpha,\beta = 1/2$, the conservation law becomes:

$$\sum_i R_{i\leftarrow j}^{(l,l+1)} = R_j^{(l+1)} \cdot \left(1 - \frac{b_j^+}{2z_j^+} - \frac{b_j^-}{2z_j^-}\right) \tag{61}$$

which has similar form to Eq (57). This alternate propagation method also allows to control manually the importance of positive and negative evidence, by choosing different factors α and β.

Once a rule for relevance propagation has been selected, the overall relevance of each neuron in the lower layer is determined by summing up the relevance coming from all upper-layer neurons in consistence with Eqs (8) and (13):

$$R_i^{(l)} = \sum_j R_{i\leftarrow j}^{(l,l+1)} \tag{62}$$

The relevance is backpropagated from one layer to another until it reaches the input pixels $x_{(d)}$, and where relevances $R_d^{(1)}$ provide the desired pixel-wise decomposition of the decision $f(x)$. The complete layer-wise relevance propagation procedure for neural networks is summarized in Algorithm 2.

Algorithm 2 Pixel-wise decomposition for neural networks
Input: $R^{(L)} = f(x)$
for $l \in \{L-1, \ldots, 1\}$ **do**
  $R_{i\leftarrow j}^{(l,l+1)}$ as in Eqs (58) or (60)
  $R_i^{(l)} = \sum_j R_{i\leftarrow j}^{(l,l+1)}$
end for
Output: $\forall d: R_d^{(1)}$

Above formulas (58) and (60) are directly applicable to layers which satisfy a certain structure. Suppose we have a neuron activation x_j from one layer which is modeled as a function of inputs from activations x_i from the preceding layer. Then layer-wise relevance propagation is directly applicable if there exists a function g_j and functions h_{ij} such that

$$x_j = g_j\left(\sum_i h_{ij}(x_i)\right) \tag{63}$$

In such a general case, the weighting terms $z_{ij} = x_i w_{ij}$ from Eq (50) have to be replaced accordingly by a function of $h_{ij}(x_i)$. A key observation is that relevance propagation is invariant against the choice of function g_j for computing relevances for the inputs x_i *conditioned on keeping the value of relevance* $\mathrm{R_j}$ *for* $\mathrm{x_j}$ *fixed*. This observation allows to deal with non-linear activation functions g_j. The function g_j does however exert influence on computing the relevance R_i by its influence on the relevance R_j for x_j. This can be explained by the fact, that the choice of g_j determines the value of x_j and thus also relevance R_j for x_j which gets assigned by the weights in the layer above.

We remark again, that even max pooling fits into this structure as a limit of generalized means, see Eq (32) for example. For structures with a higher degree of non-linearity, such as local renormalization [58, 59], Taylor approximation applied to neuron activation x_j can be used again to achieve an approximation for the structure as given in Eq (63).

Finally, it can be seen from the formulas established in this section that layer-wise relevance propagation is different from a Taylor series or partial derivatives. Unlike Taylor series, it does not require a second point other than the input image. Layer-wise application of the Taylor series can be interpreted as a generic way to achieve an approximate version of layer-wise relevance propagation. Similarly, in contrast to any methods relying on derivatives, differentiability or smoothness properties of neuron activations are not a necessary requirement for being able to define formulas which satisfy layer-wise relevance propagation. In that sense it is a more general principle.

Experiments

For Bag of Words features we show two experiments, one on an artificial but easily interpretable data set and one for on Pascal VOC images which have a high compositional complexity. For the artificial data set we apply the Taylor-type strategy for the top layer, for Pascal VOC images we apply the strategy for sum-decomposable kernels for the top layer. In both cases these strategies are combined with our definitions for arbitrary mappings for the lower layers.

For neural networks we show results also on two data sets, two sets of results on MNIST which are easy to interpret, and a second set of experiments in which we rely on a 15 layers already trained network provided as part of the Caffe open source package [60], which predicts the 1000 categories from the ILSVRC challenge. On one side, by the experiments on MNIST digits, we intend to show that we can uncover details specific to the training phase. On the other side, the results for the pre-trained network from the Caffe toolbox demonstrate, that the method works with a deep neural network out of the box and does not rely on possible tricks during the training phase.

Bag of Words features for polygons versus circles

An example of a pixel-wise decomposition for synthetic data is given in Fig 6. The training images were image tiles of size 102 × 102 pixels. An image was labeled positive if it contained at least one polygon independent of the presence of circles, and labeled negative if it contained no

TOWARDS REVERSE-ENGINEERING BLACK-BOX NEURAL NETWORKS

Seong Joon Oh, Max Augustin, Bernt Schiele, Mario Fritz
Max-Planck Institute for Informatics, Saarland Informatics Campus, Saarbrücken, Germany
{joon,maxaug,schiele,mfritz}@mpi-inf.mpg.de

arXiv:1711.01768v3 [stat.ML] 14 Feb 2018

ABSTRACT

Many deployed learned models are black boxes: given input, returns output. Internal information about the model, such as the architecture, optimisation procedure, or training data, is not disclosed explicitly as it might contain proprietary information or make the system more vulnerable. This work shows that such attributes of neural networks can be exposed from a sequence of queries. This has multiple implications. On the one hand, our work exposes the vulnerability of black-box neural networks to different types of attacks – we show that the revealed internal information helps generate more effective adversarial examples against the black box model. On the other hand, this technique can be used for better protection of private content from automatic recognition models using adversarial examples. Our paper suggests that it is actually hard to draw a line between white box and black box models. The code is available at goo.gl/MbYfsv.

1 INTRODUCTION

Black-box models take a sequence of query inputs, and return corresponding outputs, while keeping internal states such as model architecture hidden. They are deployed as black boxes usually on purpose – for protecting intellectual properties or privacy-sensitive training data. Our work aims at inferring information about the internals of black box models – ultimately turning them into white box models. Such a reverse-engineering of a black box model has many implications. On the one hand, it has legal implications to intellectual properties (IP) involving neural networks – internal information about the model can be proprietary and a key IP, and the training data may be privacy sensitive. Disclosing hidden details may also render the model more susceptible to attacks from adversaries. On the other hand, gaining information about a black-box model can be useful in other scenarios. E.g. there has been work on utilising adversarial examples for protecting private regions (e.g. faces) in photographs from automatic recognisers (Oh et al., 2017). In such scenarios, gaining more knowledge on the recognisers will increase the chance of protecting one's privacy. Either way, it is a crucial research topic to investigate the type and amount of information that can be gained from a black-box access to a model. We make a first step towards understanding the connection between white box and black box approaches – which were previously thought of as distinct classes.

We introduce the term "model attributes" to refer to various types of information about a trained neural network model. We group them into three types: (1) architecture (e.g. type of non-linear activation), (2) optimisation process (e.g. SGD or ADAM?), and (3) training data (e.g. which dataset?). We approach the problem as a standard supervised learning task *applied over models*. First, collect a diverse set of white-box models ("meta-training set") that are expected to be similar to the target black box at least to a certain extent. Then, over the collected meta-training set, train another model ("metamodel") that takes a model as input and returns the corresponding model attributes as output. Importantly, since we want to predict attributes at test time for black-box models, the only information available for attribute prediction is the query input-output pairs. As we will see in the experiments, such input-output pairs allow to predict model attributes surprisingly well.

In summary, we contribute: (1) Investigation of the type and amount of internal information about the black-box model that can be extracted from querying; (2) Novel metamodel methods that not only reason over outputs from static query inputs, but also actively optimise query inputs that can extract more information; (3) Study of factors like size of the meta-training set, quantity and quality

of queries, and the dissimilarity between the meta-training models and the test black box (generalisability); (4) Empirical verification that revealed information leads to greater susceptibility of a black-box model to an adversarial example based attack.

2 Related Work

There has been a line of work on extracting and exploiting information from black-box learned models. We first describe papers on extracting information (*model extraction* and *membership inference* attacks), and then discuss ones on attacking the network using the extracted information (*adversarial image perturbations (AIP)*).

Model extraction attacks either reconstruct the exact model parameters or build an *avatar model* that maximises the likelihood of the query input-output pairs from the target model (Tramer et al., 2016; Papernot et al., 2017). Tramer et al. (2016) have shown the efficacy of equation solving attacks and the avatar method in retrieving internal parameters of non-neural network models. Papernot et al. (2017) have also used the avatar approach with the end goal of generating adversarial examples. While the avatar approach first assumes model hyperparameters like model family (architecture) and training data, we discriminatively train a metamodel to predict those hyperparameters themselves. As such, our approach is complementary to the avatar approach.

Membership inference attacks determine if a given data sample has been included in the training data (Ateniese et al., 2015; Shokri et al., 2017). In particular, Ateniese et al. (2015) also trains a decision tree metamodel over a set of classifiers trained on different datasets. This work goes far beyond only inferring the training data by showing that even the model architecture and optimisation process can be inferred.

Using the obtained cues, one can launch more effective, focused attacks on the black box. We use *adversarial image perturbations* (AIPs) as an example of such attack. AIPs are small perturbations over the input such that the network is mislead. Research on this topic has flourished recently after it was shown that the needed amount of perturbation to completely mislead an image classifier is nearly invisible (Szegedy et al., 2014; Goodfellow et al., 2015; Moosavi-Dezfooli et al., 2017).

Most effective AIPs require gradients of the target network. Some papers proposed different ways to attack black boxes. They can be grouped into three approaches. (1) Approximate gradients by *numerical gradients* (Narodytska & Kasiviswanathan, 2017; Chen et al., 2017). The caveat is that thousands and millions of queries are needed to compute a single AIP, depending on the image size. (2) Use the *avatar approach* to train a white box network that is supposedly similar to the target (Papernot et al., 2016b;a; Hayes & Danezis, 2017). We note again that our metamodel is complementary to the avatar approach – the avatar network hyperparemters can be determined by the metamodel. (3) Exploit *transferability* of adversarial examples; it has been shown that AIPs generated against one network can also fool other networks (Moosavi-Dezfooli et al., 2017; Liu et al., 2017). Liu et al. (2017) in particular have shown that generating AIPs against an ensemble of networks make it more transferable. We show in this work that the AIPs transfer better within an architecture family (e.g. ResNet or DenseNet) than across, and that such a property can be exploited by our metamodel for generating more targetted AIPs.

3 Metamodels

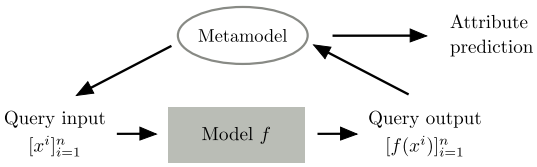


Figure 1: Overview of our approach.

We want to find out the type and amount of internal information about a black-box model that can be revealed from a sequence of queries. We approach this by first building metamodels for predicting model attributes, and then evaluating their performance on black-box models. Our main approach, metamodel, is described in figure 1. In a nutshell, the metamodel is a classifier of classifiers. Specifically, The metamodel submits n query inputs $\left[x^i\right]_{i=1}^n$ to a black box model f; the metamodel takes corresponding model outputs $\left[f(x^i)\right]_{i=1}^n$ as an input, and returns predicted model attributes as output. As we will describe in detail, the metamodel not only learns to infer model

attributes from query outputs from a static set of inputs, but also searches for query inputs that are designed to extract greater amount of information from the target models.

In this section, our main methods are introduced in the context of MNIST digit classifiers. While MNIST classifiers are not fully representative of *generic* learned models, they have a computational edge: it takes only five minutes to train each of them with reasonable performance. We could thus prepare a diverse set of 11k MNIST classifiers within 40 GPU days for the meta-training and evaluation of our metamodels. We stress, however, that the proposed approach is generic with respect to the task, data, and the type of models. We also focus on 12 model attributes (table 1) that cover hyperparameters for common neural network MNIST classifiers, but again the range of predictable attributes are not confined to this list.

3.1 COLLECTING A DATASET OF CLASSIFIERS

We need a dataset of classifiers to train and evaluate metamodels. We explain how MNIST-NETS has been constructed, a dataset of 11k MNIST digit classifiers; the procedure is task and data generic.

BASE NETWORK SKELETON

Every model in MNIST-NETS shares the same convnet skeleton architecture: "N conv blocks $\rightarrow$ M fc blocks $\rightarrow$ 1 linear classifier". Each conv block has the following structure: "ks $\times$ ks convolution $\rightarrow$ optional 2×2 max-pooling $\rightarrow$ non-linear activation", where ks (kernel size) and the activation type are to be chosen. Each fc block has the structure: "'linear mapping $\rightarrow$ non-linear activation $\rightarrow$ optional dropout" This convnet structure already covers many LeNet (LeCun et al., 1998) variants, one of the best performing architectures on MNIST[1].

INCREASING DIVERSITY

In order to learn generalisable features, the metamodel needs to be trained over a diverse set of models. The base architecture described above already has several free parameters like the number of layers (N and M), the existence of dropout or max-pooling layers, or the type of non-linear activation.

Table 1: MNIST classifier attributes. *Italicised* attributes are derived from other attributes.

	Code	Attribute	Values
Architecture	act	Activation	ReLU, PReLU, ELU, Tanh
	drop	Dropout	Yes, No
	pool	Max pooling	Yes, No
	ks	Conv ker. size	3, 5
	#conv	#Conv layers	2, 3, 4
	#fc	#FC layers	2, 3, 4
	#par	*#Parameters*	$2^{14}, \cdots, 2^{21}$
	ens	Ensemble	Yes, No
Opt.	alg	Algorithm	SGD, ADAM, RMSprop
	bs	Batch size	64, 128, 256
Data	split	Data split	All_0, $\text{Half}_{0/1}$, $\text{Quarter}_{0/1/2/3}$
	size	*Data size*	All, Half, Quarter

Apart from the architectural hyperparameters, we increase diversity along two more axes – optimisation process and the training data. Along the optimisation axis, we vary optimisation algorithm (SGD, ADAM, or RMSprop) and the training batch size (64, 128, 256). We also consider training MNIST classifiers on either on the entire MNIST training set (All_0, 60k), one of the two disjoint halves ($\text{Half}_{0/1}$, 30k), or one of the four disjoint quarters ($\text{Quarter}_{0/1/2/3}$, 15k).

See table 1 for the comprehensive list of 12 model attributes altered in MNIST-NETS. The number of trainable parameters (#par) and the training data size (size) are not directly controlled but derived from the other attributes. We also augment MNIST-NETS with ensembles of classifiers (ens), whose procedure will be described later.

SAMPLING AND TRAINING

The number of all possible combinations of controllable options in table 1 is $18,144$. We also select random seeds that control the initialisation and training data shuffling from $\{0, \cdots, 999\}$, resulting in $18,144,000$ unique models. Training such a large number of models is intractable; we have sampled (without replacement) and trained $10,000$ of them. All the models have been trained with

[1] http://yann.lecun.com/exdb/mnist/

learning rate 0.1 and momentum 0.5 for 100 epochs. It takes around 5 minutes to train each model on a GPU machine (GeForce GTX TITAN); training of 10k classifiers has taken 40 GPU days.

PRUNING AND AUGMENTING

In order to make sure that MNIST-NETS realistically represents commonly used MNIST classifiers, we have pruned low-performance classifiers (validation accuracy$< 98\%$), resulting in $8,582$ classifiers. Ensembles of trained classifiers have been constructed by grouping the identical classifiers (modulo random seed). Given t identical ones, we have augmented MNIST-NETS with $2, \cdots, t$ combinations. The ensemble augmentation has resulted in $11,282$ final models. See appendix table 6 for statistics of attributes – due to large sample size all the attributes are evenly covered.

TRAIN-EVAL SPLITS

Attribute prediction can get arbitrarily easy by including the black-box model (or similar ones) in the meta-training set. We introduce multiple splits of MNIST-NETS with varying requirements on generalization. Unless stated otherwise, every split has $5,000$ training (meta-training), $1,000$ testing (black box), and $5,282$ leftover models.

The Random (R) split randomly (uniform weights) assigns training and test splits, respectively. Under the R split, the training and test models come from the same distribution. We introduce harder Extrapolation (E) splits. We separate a few attributes between the training and test splits. They are designed to simulate more difficult domain gaps when the meta-training models are significantly different from the black box. Specific examples of E splits will be shown in §4.

3.2 METAMODEL METHODS

The metamodel predicts the attribute of a black-box model g in the test split by submitting n query inputs and observing the outputs. It is trained over meta-training models f in the training split ($f \sim \mathcal{F}$). We propose three approaches for the metamodels – we collectively name them kennen[2]. See figure 2 for an overview.

KENNEN-O: REASON OVER OUTPUT

kennen-o first selects a fixed set of queries $[x^i]_{i=1\cdots n}$ from a dataset. Both during training and testing, always these queries are submitted. kennen-o learns a classifier m_θ to map from the order-sensitively concatenated n query outputs, $[f(x^i)]_{i=1\cdots n}$ ($n \times 10$ dim for MNIST), to the simultaneous prediction of 12 attributes in f. The training objective is:

$$\min_\theta \mathop{\mathbb{E}}_{f\sim\mathcal{F}} \left[\sum_{a=1}^{12} \mathcal{L}\left(m_\theta^a\left([f(x^i)]_{i=1}^n\right), y^a\right) \right] \quad (1)$$

where $\mathcal{F}$ is the distribution of meta-training models, y^a is the ground truth label of attribute a, and $\mathcal{L}$ is the cross-entropy loss. With the learned parameter $\tilde{\theta}$, $m_{\tilde{\theta}}^a\left([g(x^i)]_{i=1}^n\right)$ gives the prediction of attribute a for the black box g.

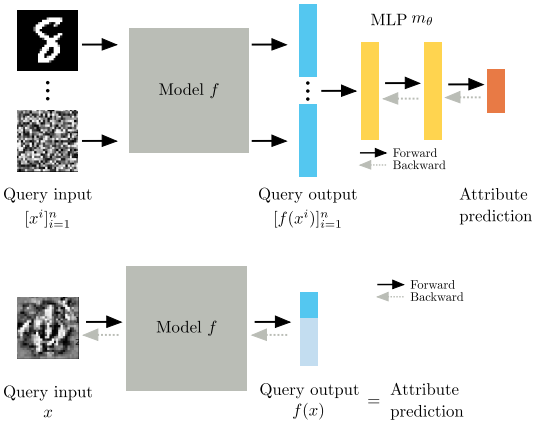


Figure 2: Training procedure for metamodels kennen-o (top) and kennen-i (bottom).

In our experiments, we model the classifier m_θ via multilayer perceptron (MLP) with two hidden layers with 1000 hidden units. The last layer consists of 12 parallel linear layers for a simultaneous prediction of the attributes. In our preliminary experiments, MLP has performed better than the linear classifiers. The optimisation problem in equation 1 is solved via SGD by approximating the expectation over $f \sim \mathbb{F}$ by an empirical sum over the training split classifiers for 200 epochs.

[2] *kennen* means "to know" in German, and "to dig out" in Korean.

For query inputs, we have used a random subset of n images from the validation set (both for MNIST and ImageNet experiments). The performance is not sensitive to the choice of queries (see appendix §C). Next methods (`kennen-i/io`) describe how to actively craft query inputs, potentially outside the natural image distribution.

Note that `kennen-o` can be applied to any type of model (e.g. non-neural networks) with any output structure, as long as the output can be embedded in an Euclidean space. We will show that this method can effectively extract information from f even if the output is a top-k ranking.

KENNEN-I: CRAFT INPUT

`kennen-i` crafts a *single* query input $\tilde{x}$ over the meta-training models that is trained to repurpose a digit classifier f into a model attribute classifier for a *single* attribute a. The crafted input drives the classifier to leak internal information via digit prediction. The learned input is submitted to the test black-box model g, and the attribute is predicted by reading off its digit prediction $g(\tilde{x})$. For example, `kennen-i` for max-pooling layer prediction crafts an input x that is predicted as "1" for generic MNIST digit classifiers with max-pooling layers and "0" for ones without. See figure 3 for visual examples.

We describe in detail how `kennen-i` learns this input. The training objective is:

$$\min_{x\text{: image}} \mathop{\mathbb{E}}_{f\sim\mathcal{F}} \left[\mathcal{L}\left(f(x), y^a\right)\right] \tag{2}$$

where $f(x)$ is the 10-dimensional output of the digit classifier f. The condition x : image ensures the input stays a valid image $x \in [0,1]^D$ with image dimension D. The loss $\mathcal{L}$, together with the attribute label y^a of f, guides the digit prediction $f(x)$ to reveal the attribute a instead. Note that the optimisation problem is identical to the training of digit classifiers except that the ground truth is the attribute label rather than the digit label, that the loss is averaged over the models instead of the images, and that the input x instead of the model f is optimised. With the learned query input $\tilde{x}$, the attribute for the black box g is predicted by $g(\tilde{x})$. In particular, we do not use gradient information from g.

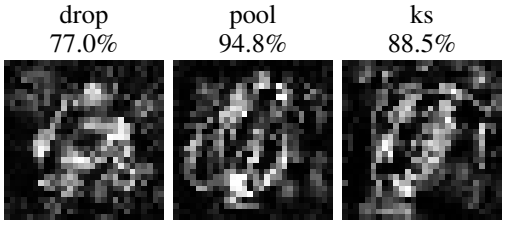


Figure 3: Inputs designed to extract internal details from MNIST digit classifiers. E.g. feeding the middle image reveals the existence of a max-pooling layer with 94.8% chance.

We initialise x with a random sample from the MNIST validation set (random noise or uniform gray initialisation gives similar performances), and run SGD for 200 epochs. For each iteration x is truncated back to $[0,1]^D$ to enforce the constraint.

While being simple and effective, `kennen-i` can only predict a single attribute at a time, and cannot predict attributes with more than 10 classes (for digit classifiers). `kennen-io` introduced below overcomes these limitations. `kennen-i` may also be unrealistic when the exploration needs to be stealthy: it submits unnatural images to the system. Also unlike `kennen-o`, `kennen-i` requires end-to-end differentiability of the training models $f \sim \mathcal{F}$, although it still requires only black-box access to test models g.

KENNEN-IO: COMBINED APPROACH

We overcome the drawbacks of `kennen-i` that it can only predict one attribute at a time and that the number of predictable classes by attaching an additional interpretation module on top of the output. Our final method `kennen-io` combines `kennen-i` and `kennen-o` approaches: both input generator and output interpreters are used. Being able to reason over multiple query outputs via MLP layers, `kennen-io` supports the optimisation of multiple query inputs as well.

Specifically, the `kennen-io` training objective is given by:

$$\min_{[x^i]_{i=1}^n\text{: images}} \min_{\theta} \mathop{\mathbb{E}}_{f\sim\mathcal{F}} \left[\sum_{a=1}^{12} \mathcal{L}\left(m_\theta^a\left([f(x^i)]_{i=1}^n\right), y^a\right)\right]. \tag{3}$$

Note that the formulation is identical to that for `kennen-o` (equation 1), except that the second minimisation problem regarding the query inputs is added. With learned parameters $\tilde{\theta}$ and $[\tilde{x}^i]_{i=1}^n$,

Table 2: Comparison of metamodel methods. See table 1 for the full names of attributes. 100 queries are used for every method below, except for `kennen-i` which uses a single query. The "Output" column shows the output representation: "prob" (vector of probabilities for each digit class), "ranking" (a sorted list of digits according to their likelihood), "top-1" (most likely digit), or "bottom-1" (least likely digit).

		architecture								optim		data		
Method	Output	act	drop	pool	ks	#conv	#fc	#par	ens	alg	bs	size	split	avg
Chance	-	25.0	50.0	50.0	50.0	33.3	33.3	12.5	50.0	33.3	33.3	33.3	14.3	34.9
`kennen-o`	prob	80.6	94.6	94.9	84.6	67.1	77.3	41.7	54.0	71.8	50.4	73.8	90.0	73.4
`kennen-o`	ranking	63.7	93.8	90.8	80.0	63.0	73.7	44.1	**62.4**	65.3	47.0	66.2	86.6	69.7
`kennen-o`	bottom-1	48.6	80.0	73.6	64.0	48.9	63.1	28.7	52.8	53.6	41.9	45.9	51.4	54.4
`kennen-o`	top-1	31.2	56.9	58.8	49.9	38.9	33.7	19.6	50.0	36.1	35.3	33.3	30.7	39.5
`kennen-i`	top-1	43.5	77.0	94.8	88.5	54.5	41.0	32.3	46.5	45.7	37.0	42.6	29.3	52.7
`kennen-io`	score	**88.4**	**95.8**	**99.5**	**97.7**	**80.3**	**80.2**	**45.2**	60.2	**79.3**	**54.3**	**84.8**	**95.6**	**80.1**

the attribute a for the black box g is predicted by $m^a_{\tilde{\theta}}\left([g(\tilde{x}^i)]_{i=1}^n\right)$. Again, we require end-to-end differentiability of meta-training models f, but only the black-box access for the test model g.

To improve stability against covariate shift, we initialise m_θ with `kennen-o` for 200 epochs. Afterwards, gradient updates of $[x^i]_{i=1}^n$ and θ alternate every 50 epochs, for 200 additional epochs.

4 Reverse-engineering Black-Box MNIST Digit Classifiers

We have introduced a procedure for constructing a dataset of classifiers (`MNIST-NETS`) as well as novel metamodels (`kennen` variants) that learn to extract information from black-box classifiers. In this section, we evaluate the ability of `kennen` to extract information from black-box MNIST digit classifiers. We measure the *class-balanced* attribute prediction accuracy for each attribute a in the list of 12 attributes in table 1.

Attribute prediction

See table 2 for the main results of our metamodels, `kennen-o/i/io`, on the Random split. Unless stated otherwise, metamodels are trained with $5,000$ training split classifiers.

Given $n = 100$ queries with probability output, `kennen-o` already performs far above the random chance in predicting 12 diverse attributes (73.4% versus 34.9% on average); neural network output indeed contains rich information about the black box. In particular, the presence of dropout (94.6%) or max-pooling (94.9%) has been predicted with high precision. As we will see in §4.3, outputs of networks trained with dropout layers form clusters, explaining the good prediction performance.

It is surprising that optimisation details like algorithm (71.8%) and batch size (50.4%) can also be predicted well above the random chance (33.3% for both). We observe that the training data attributes are also predicted with high accuracy (71.8% and 90.0% for size and split).

Comparing methods `kennen-o/i/io`

Table 2 shows the comparison of `kennen-o/i/io`. `kennen-i` has a relatively low performance (average 52.7%), but `kennen-i` relies on a cheap resource: 1 query with single-label output. `kennen-i` is also performant at predicting the kernel size (88.5%) and pooling (94.8%), attributes that are closely linked to spatial structure of the input. We conjecture `kennen-i` is relatively effective for such attributes. `kennen-io` is superior to `kennen-o/i` for all the attributes with average accuracy 80.1%.

4.1 Factor analysis

We examine potential factors that contribute to the successful prediction of black box internal attributes. We measure the prediction accuracy of our metamodels as we vary (1) the number of meta-training models, (2) the number of queries, and (3) the quality of query output.

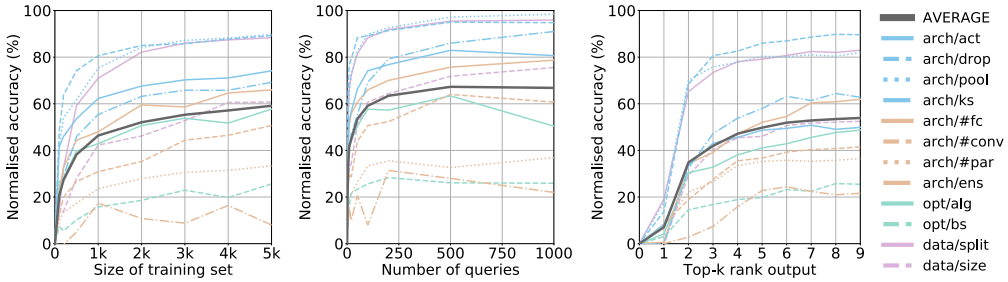


Figure 4: `kennen-o` performance of against the size of meta-training set (left), number of queries (middle), and quality of queries (right). Unless stated otherwise, we use 100 probability outputs and 5k models to train `kennen-o`. Each curve is linearly scaled such that random chance (0 training data, 0 query, or top-0) performs 0%, and the perfect predictor performs 100%.

NUMBER OF TRAINING MODELS

We have trained `kennen-o` with different number of the meta-training classifiers, ranging from 100 to $5,000$. See figure 4 (left) for the trend. We observe a diminishing return, but also that the performance has not saturated – collecting larger meta-training set will improve the performance.

NUMBER OF QUERIES

See figure 4 (middle) for the `kennen-o` performance against the number of queries with probability output. The average performance saturates after ~ 500 queries. On the other hand, with only ~ 100 queries, we already retrieve ample information about the neural network.

QUALITY OF OUTPUT

Many black-box models return top-k ranking (e.g. Facebook face recogniser), or single-label output. We represent top-k ranking outputs by assigning exponentially decaying probabilities up to k digits and a small probability ϵ to the remaining.

See table 2 for the `kennen-o` performance comparison among 100 probability, top-10 ranking, bottom-1, and top-1 outputs, with average accuracies 73.4%, 69.7%, 54.4%, and 39.5%, respectively. While performance drops with coarser outputs, when compared to random chance (34.9%), 100 single-label bottom-1 outputs already leak a great amount of information about the black box (54.4%). It is also notable that bottom-1 outputs contain much more information than do the top-1 outputs; note that for high-performance classifiers top-1 predictions are rather uniform across models and thus have much less freedom to leak auxiliary information. Figure 4 (right) shows the interpolation from top-1 to top-10 (i.e. top-9) ranking. We observe from the jump at $k = 2$ that the second likely predictions (top-2) contain far more information than the most likely ones (top-1). For $k \geq 3$, each additional output label exhibits a diminishing return.

4.2 WHAT IF THE BLACK-BOX IS QUITE DIFFERENT FROM META-TRAINING MODELS?

So far we have seen results on the Random (R) split. In realistic scenarios, the meta-training model distribution may not be fully covering possible black box models. We show how damaging such a scenario is through Extrapolation (E) split experiments.

EVALUATION

E-splits split the training and testing models based on one or more attributes (§3.1). For example, we may assign shallower models (#layers ≤ 10) to the training split and deeper ones (#layers ¿10) to the testing split. In this example, we refer to #layers as the *splitting attribute*. Since for an E-split, some classes of the splitting attributes have zero training examples, we only evaluate the prediction accuracies over the non-splitting attributes. When the set of splitting attributes is $\tilde{A}$, a subset of the entire attribute set A, we define *E-split accuracy* or E.Acc$(\tilde{A})$ to be the mean prediction accuracy over the non-splitting attributes $A \setminus \tilde{A}$. For easier comparison, we report the *normalised accuracy*

(N.Acc) that shows the how much percentage of the R-split accuracy is achieved in the E-split setup on the non-splitting attributes $A \setminus \tilde{A}$. Specifically:

$$\text{N.Acc}(\tilde{A}) = \frac{\text{E.Acc}(\tilde{A}) - \text{Chance}(\tilde{A})}{\text{R.Acc}(\tilde{A}) - \text{Chance}(\tilde{A})} \times 100\% \tag{4}$$

where $\text{R.Acc}(\tilde{A})$ and $\text{Chance}(\tilde{A})$ are the means of the R-split and Chance-level accuracies over $A \setminus \tilde{A}$. Note that N.Acc is 100% if the E-split performance is at the level of R-split and 0% if it is at chance level.

RESULTS

The normalised accuracies for R-split and multiple E-splits are presented in table 3. We consider three axes of choices of splitting attributes for the E-split: architecture (#conv and #fc), optimisation (alg and bs), and data (size). For example, "E-#conv-#fc" row presents results when metamodel is trained on shallower nets (2 or 3 conv/fc layers each) compared to the test black box model (4 conv and fc layers each).

Not surprisingly, E-split performances are lower than R-split ones (N.Acc $< 100\%$); it is advisable to cover all the expected black-box attributes during meta-training. Nonetheless, E-split performances of `kennen-io` are still far above the chance level (N.Acc $\geq 70\% \gg 0\%$); failing to cover a few attributes during meta-training is not too damaging.

Table 3: Normalised accuracies (see text) of `kennen-o` and `kennen-io` on R and E splits. We denote E-split with splitting attributes *attr1* and *attr2* as "E-*attr1*-*attr2*". Splitting criteria are also shown. When there are two splitting attributes, the first attribute inherits the previous row criteria.

			kennen-	
Split	Train	Test	o	io
R	-	-	100	100
E-#conv	2,3	4	87.5	92.0
E-#conv-#fc	2,3	4	77.1	80.7
E-alg	SGD,ADAM	RMSprop	83.0	88.5
E-alg-bs	64,128	256	64.2	70.0
E-split	$\text{Quarter}_{0/1}$	$\text{Quarter}_{2/3}$	83.5	89.3
E-size	Quarter	Half,All	81.7	86.8
Chance	-	-	0.0	0.0

Comparing `kennen-o` and `kennen-io` for their generalisability, we observe that `kennen-io` consistently outperforms `kennen-o` under severe extrapolation (around 5 pp better N.Acc). It is left as a future work to investigate the intriguing fact that utilising out-of-domain query inputs improves the generalisation of metamodel.

4.3 WHY AND HOW DOES METAMODEL WORK?

It is surprising that metamodels can extract inner details with great precision and generalisability. This section provides a glimpse of *why* and *how* this is possible via metamodel input and output analyses. Full answers to those questions is beyond the scope of the paper.

METAMODEL INPUT (T-SNE)

We analyse the inputs to our metamodels (i.e. query outputs from black-box models) to convince ourselves that the inputs do contain discriminative features for model attributes. As the input is high dimensional (1000 when the number of queries is $n = 100$), we use the t-SNE (van der Maaten & Hinton, Nov 2008) visualisation method. Roughly speaking, t-SNE embeds high dimensional data points onto the 2-dimensional plane such that the pairwise distances are best respected. We then colour-code the embedded data points according to the model attributes. Clusters of same-coloured points indicate highly discriminative features.

The visualisation of input data points are shown in Appendix figures 9 and 10 for `kennen-o` and `kennen-io`, respectively. For experimental details, see Appendix §D. In the case of `kennen-o`, we observe that some attributes form clear clusters in the input space – e.g. Tanh in act, binary dropout attribute, and RMSprop in alg. For the other attributes, however, it seems that the clusters are too complicated to be represented in a 2-dimensional space. For `kennen-io` (figure 10), we observe improved clusters for pool and ks. By submitting crafted query inputs, `kennen-io` induces query outputs to be better clustered, increasing the chance of successful prediction.

METAMODEL OUTPUT (CONFUSION MATRIX)

We show confusion matrices of `kennen-o/io` to analyse the failure modes. See Appendix figures 11 and 12. For `kennen-o` and `kennen-io` alike, we observe that the confusion occurs more frequently with similar classes. For attributes #conv and #fc, more confusion occurs between $(2, 3)$ or $(3, 4)$ than between $(2, 4)$. A similar trend is observed for #par and bs. This is a strong indication that (1) there exists semantic attribute information in the neural network outputs (e.g. number of layers, parameters, or size of training batch) and (2) the metamodels learn semantic information that can generalise, as opposed to merely relying on artifacts. This observation agrees with a conclusion of the extrapolation experiments in §4.2: the metamodels generalise.

Compared to those of `kennen-o`, `kennen-io` confusion matrices exhibit greater concentration of masses both on the correct class (diagonals) and among similar attribute classes (1-off diagonals for #conv, #fc, #par, bs, and size). The former re-confirms the greater accuracy, while the latter indicates the improved ability to extract more semantic and generalisable features from the query outputs. This, again, agrees with §4.2: `kennen-io` generalises better than `kennen-o`.

4.4 DISCUSSION

We have verified through our novel `kennen` metamodels that black-box access to a neural network exposes much internal information. We have shown that only 100 single-label outputs already reveals a great deal about a black box. When the black-box classifier is quite different from the meta-training classifiers, the performance of our best metamodel – `kennen-io`– decreases; however, the prediction accuracy for black box internal information is still surprisingly high.

5 REVERSE-ENGINEERING AND ATTACKING IMAGENET CLASSIFIERS

While MNIST experiments are computationally cheap and a massive number of controlled experiments is possible, we provide additional ImageNet experiments for practical implications on realistic image classifiers. In this section, we use `kennen-o` introduced in §3 to predict a single attribute of black-box ImageNet classifiers – the architecture family (e.g. ResNet or VGG?). In this section, we go a step further to use the extracted information to attack black boxes with adversarial examples.

5.1 DATASET OF IMAGENET CLASSIFIERS

It is computationally prohibitive to train $O(10k)$ ImageNet classifiers from scratch as in the previous section. We have resorted to 19 PyTorch[3] pretrained ImageNet classifiers. The 19 classifiers come from five families: **S**queezenet, **V**GG, VGG-**B**atchNorm, **R**esNet, and **D**enseNet, each with 2, 4, 4, 5, and 4 variants, respectively (Iandola et al., 2016; Simonyan & Zisserman, 2015; Ioffe & Szegedy, 2015; He et al., 2016; Huang et al., 2017). See Appendix table 7 for the the summary of the 19 classifiers. We observe both large intra-family diversity and small inter-family separability in terms of #layers, #parameters, and performances. The family prediction task is not as trivial as e.g. simply inferring the performance.

5.2 CLASSIFIER FAMILY PREDICTION

We predict the classifier family (S, V, B, R, D) from the black-box query output, using the method `kennen-o`, with the same MLP architecture (§3). `kennen-i` and `kennen-io` have not been used for computational reasons, but can also be used in principle. We conduct 10 cross validations (random sampling of single test network from each family) for evaluation. We also perform 10 random sampling of the queries from ImageNet validation set. In total 100 random tries are averaged.

Results: compared to the random chance (20.0%), 100 queries result in high `kennen-o` performance (90.4%). With $1,000$ queries, the prediction performance is even 94.8%.

[3] https://github.com/pytorch

5.3 ATTACKING IMAGENET CLASSIFIERS

In this section we attack ImageNet classifiers with adversarial image perturbations (AIPs). We show that the knowledge about the black box architecture family makes the attack more effective.

ADVERSARIAL IMAGE PERTURBATION (AIP)

AIPs are carefully crafted additive perturbations on the input image for the purpose of misleading the target model to predict wrong labels (Goodfellow et al., 2015). Among variants of AIPs, we use efficient and robust GAMAN (Oh et al., 2017). See appendix figure 7 for examples of AIPs; the perturbation is nearly invisible.

TRANSFERABILITY OF AIPS

Typical AIP algorithms require gradients from the target network, which is not available for a black box. Mainly three approaches for generating AIPs against black boxes have been proposed: (1) numerical gradient, (2) avatar network, or (3) transferability. We show that our metamodel strengthens the transferability based attack.

We hypothesize and empirically show that AIPs transfer better within the architecture family than across. Using this property, we first predict the family of the black box (e.g. ResNet), and then generate AIPs against a few instances in the family (e.g. ResNet101, ResNet152). The generation of AIPs against multiple targets has been proposed by Liu et al. (2017), but we are the first to systemically show that AIPs generalise better within a family when they are generated against multiple instances from the same family.

Table 4: Transferability of adversarial examples within and across families. We report *misclassification rates*.

	Target family				
Gen	S	V	B	R	D
Clean	38	32	28	30	29
S	64	49	45	39	35
V	62	96	96	57	52
B	50	85	95	47	44
R	64	72	78	87	77
D	58	63	70	76	90
Ens	70	93	93	75	80

We first verify our hypothesis that AIPs transfer better within a family. Within-family: we do a leave-one-out cross validation – generate AIPs using all but one instances of the family and test on the holdout. Not using the exact test black box, this gives a lower bound on the within-family performance. Across-family: still leave out one random instance from the generating family to match the generating set size with the within-family cases. We also include the use-all case (Ens): generate AIPs with one network from *each* family.

See table 4 for the results. We report the *misclassification rate*, defined as $100-$top-1 accuracy, on 100 random ImageNet validation images. We observe that the within-family performances dominate the across-family ones (diagonal entries versus the others in each row); if the target black box family is identified, one can generate more effective AIPs. Finally, trying to target all network ("Ens") is not as effective as focusing resources (diagonal entries).

METAMODEL ENABLES MORE EFFECTIVE ATTACKS

We empirically show that the reverse-engineering enables more effective attacks. We consider multiple scenarios. "White box" means the target model is fully known, and the AIP is generated specifically for this model. "Black box" means the exact target is unknown, but we make a distinction when the family is known ("Family black box").

See table 5 for the misclassification rates (MC) in different scenarios. When the target is fully specified (white box), MC is 100%. When neither the exact target nor the family is known, AIPs are generated against multiple families (82.2%). When the reverse-engineering takes place, and AIPs are generated over the predicted family, attacks become more effective (85.7%). We almost reach the family-oracle case (86.2%).

Table 5: Black-box ImageNet classifier misclassification rates (MC) for different approaches.

Scenario	Generating nets	MC(%)
White box	Single white box	100.0
Family black box	GT family	86.2
Black box whitened	**Predicted family**	**85.7**
Black box	Multiple families	82.2

5.4 DISCUSSION

Our metamodel can predict architecture families for ImageNet classifiers with high accuracy. We additionally show that this reverse-engineering enables more focused attack on black-boxes.

6 CONCLUSION

We have presented first results on the inference of diverse neural network attributes from a sequence of input-output queries. Our novel metamodel methods, `kennen`, can successfully predict attributes related not only to the architecture but also to training hyperparameters (optimisation algorithm and dataset) even in difficult scenarios (e.g. single-label output, or a distribution gap between the meta-training models and the target black box). We have additionally shown in ImageNet experiments that reverse-engineering a black box makes it more vulnerable to adversarial examples.

ACKNOWLEDGMENTS

This research was supported by the German Research Foundation (DFG CRC 1223). We thank Seong Ah Choi for her help with the method names, graphics, and colour palettes.

REFERENCES

Giuseppe Ateniese, Giovanni Felici, Liugi V. Mancini, Angelo Spognardi, Antonio Villani, and Domenico Vitali. Hacking smart machines with smarter ones: How to extract meaningful data from machine learning classifiers. In *IJSN*, 2015.

Pin-Yu Chen, Huan Zhang, Yash Sharma, Jinfeng Yi, and Cho-Jui Hsieh. Zoo: Zeroth order optimization based black-box attacks to deep neural networks without training substitute models. In *ACMCCS-W*, 2017.

Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *ICLR*, 2015.

Jamie Hayes and George Danezis. Machine learning as an adversarial service: Learning black-box adversarial examples. 2017.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5mb model size. *arXiv*, 2016.

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

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 1998.

Yanpei Liu, Xinyun Chen, Chang Liu, and Dawn Song. Delving into transferable adversarial examples and black-box attacks. In *ICLR*, 2017.

Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, Omar Fawzi, and Pascal Frossard. Universal adversarial perturbations. In *CVPR*, 2017.

Nina Narodytska and Shiva Prasad Kasiviswanathan. Simple black-box adversarial perturbations for deep networks. In *CVPRW*, 2017.

S. J. Oh, Mario Fritz, and Bernt Schiele. Adversarial image perturbation for privacy protection a game theory perspective. In *ICCV*, 2017.

Nicolas Papernot, Patrick McDaniel, and Ian Goodfellow. Transferability in machine learning: from phenomena to black-box attacks using adversarial samples. *arXiv*, 2016a.

Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z. Berkay Celik, and Anathram Swami. Practical black-box attacks against deep learning systems using adversarial examples. 2016b.

Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. Practical black-box attacks against deep learning systems using adversarial examples. In *ASIACCS*, 2017.

Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *SP*, 2017.

K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2015.

Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *ICLR*, 2014.

Florian Tramer, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction apis. In *USENIX*, 2016.

L.J.P van der Maaten and G.E. Hinton. Visualizing high-dimensional data using t-sne. *Journal of Machine Learning Research*, 9: 25792605, Nov 2008.

APPENDIX

A MNIST-NETS STATISTICS

We show the statistics of MNIST-NETS, our dataset of MNIST classifiers, in table 6.

B MORE KENNEN-IO RESULTS

We complement the kennen-o results in the main paper (figure 4) with kennen-io results. See figure 5. Similarly for kennen-o, kennen-io shows a diminishing return as the number of training models and the number of queries increase. While the performance saturates with $1,000$ queries, it does not fully saturate with $5,000$ training samples.

C ON FINDING THE OPTIMAL SET OF QUERIES

kennen-o selects a random set of queries from MNIST validation set (§3.2). We measure the sensitivity of kennen-o performance with respect to the choice of queries, and discuss the possibility to optimise the set of queries.

With 1, 10, or 100 queries, we have trained kennen-o with 100 independent samples of query sets. The mean and standard deviations are shown in figure 6. The sensitivity is greater for smaller number of queries, but still minute (1.2 pp standard deviation).

Instead of solving the combinatorial problem of finding the optimal set of query inputs from a dataset, we have proposed kennen-io that efficiently solves a continuous optimisation problem to find a set of query inputs from the entire input space. We have compared kennen-io against kennen-o with multiple query samples in figure 6. We observe that kennen-io is better than kennen-o with all 100 query set samples at each level.

We remark that there exists a trade-off between detectability and effectiveness of exploration. While kennen-io extracts information from target model more effectively, it increases the detectability of attack by submitting out-of-domain inputs. If it is possible to optimise or sample the set of natural queries from a dataset or distribution of natural inputs, it will be a strong attack; developing such a method would be an interesting future work.

D T-SNE VISUALISATION OF METAMODEL INPUTS

We describe the detailed procedure for the metamodel input visualisation experiment (discussed in §4.3). First, 1000 test-split (Random split) black-box models are collected. For each model, 100 query images are passed (sampled at random from MNIST validation set), resulting in 100×10 dimensional input data points. We have used t-SNE(van der Maaten & Hinton, Nov 2008) to embed the data points onto the 2-dimensional plane. Each data point is coloured according to each attribute class. The results for kennen-o and kennen-io are shown in figures 9 and 10. Since t-SNE is sensitive to initialisation, we have run the embedding ten times with different random initialisations; the qualitative observations are largely identical.

E VISUAL EXAMPLES OF AIPS

In this section, we show examples of AIPs. See figure 7 for the examples of AIPs and the perturbed images. The perturbation is nearly invisible to human eyes. We have also generated AIPs with respect to a diverse set of architecture families (S, V, B, R, D, SVBRD) at multiple L_2 norm levels. See figure 8; the same image results in a diverse set of patterns depending on the architecture family.

Table 6: Distribution of attributes in `MNIST-NETS`, and attribute-wise classification performance (on MNIST validation set). Observe that the attributes are evenly distributed and the corresponding classification accuracies also do not correlate much with the attributes. We thus make sure that the classification accuracy alone cannot be a strong cue for predicting attributes.

	arch/act				arch/drop		arch/pool		arch/ks		arch/#conv			arch/#fc		
	Tanh	PReLU	ReLU	ELU	Yes	No	Yes	No	5	3	2	3	4	2	3	4
Ratio	24.8	24.9	25.3	25.1	49.8	50.3	49.9	50.2	50.3	49.7	34.0	33.4	32.7	33.1	33.5	33.4
max	99.4	99.4	99.5	99.4	99.5	99.4	99.4	99.5	99.5	99.4	99.4	99.4	99.5	99.4	99.4	99.5
median	98.6	98.7	98.7	98.7	98.7	98.6	98.7	98.5	98.7	98.6	98.6	98.7	98.7	98.7	98.6	98.6
mean	98.6	98.7	98.7	98.7	98.7	98.6	98.7	98.6	98.7	98.6	98.6	98.7	98.7	98.7	98.6	98.6
min	98.0	98.0	98.0	98.0	98.0	98.0	98.0	98.0	98.0	98.0	98.0	98.0	98.0	98.0	98.0	98.0

	opt/alg			opt/bs			data/size		
	RMSprop	ADAM	SGD	64	128	256	all	half	quarter
Ratio	33.8	32.5	33.7	32.9	33.6	33.7	14.8	28.5	56.8
max	99.2	99.4	99.5	99.3	99.4	99.5	99.5	99.3	99.1
median	98.6	98.7	98.7	98.6	98.7	98.7	99.0	98.8	98.5
mean	98.6	98.7	98.7	98.6	98.7	98.6	98.9	98.8	98.5
min	98.0	98.0	98.0	98.0	98.0	98.0	98.0	98.0	98.0

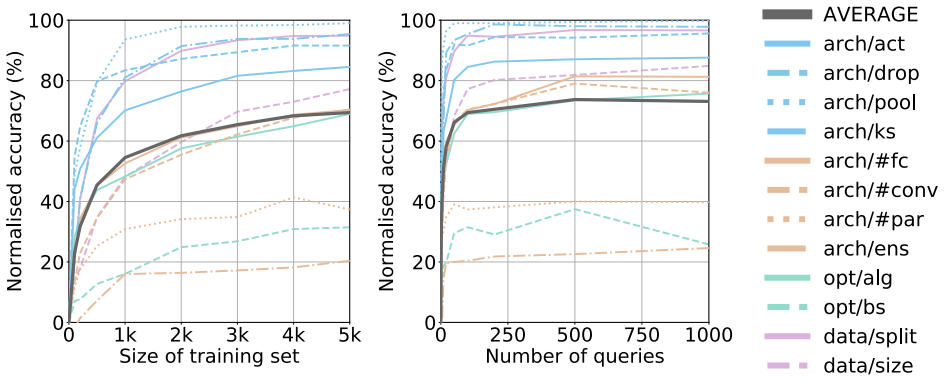


Figure 5: Performance of `kennen-io` with different number of queries (Left) and size of training set (Right). The curves are linearly scaled per attribute such that random chance performs 0%, and perfect predictor performs 100%.

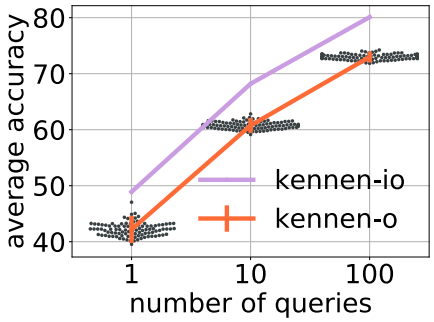


Figure 6: `kennen-o/io` performance at different number of queries. `kennen-o` is shown with 100 independent query samples per level (black dots) – the dots are spread horizontally for visualisation purpose. Their mean (curve) and ± 2 standard deviations (error bars) are also shown.

Table 7: Details of ImageNet classifiers. We describe each family **S**queezenet, **V**GG, VGG-**B**atchNorm, **R**esNet, and **D**enseNet verbally, and show key model statistics for each member in the family. We observe intra-family diversity (e.g. R) and inter-family similarity (e.g. between V and B) in terms of the top-5 validation error and the number of trainable parameters.

	S (2016)		V (2014)				B (2015)				R (2015)					D (2016)			
Description	Lightweight convnet		Conv layers followed by fc layers				VGG with batch normalisation				Very deep convnet with residual connections					ResNet with dense residual connections			
Members	v1.0	v1.1	11	13	16	19	11	13	16	19	18	34	50	101	152	121	161	169	201
#layers	26	26	11	13	16	19	11	13	16	19	21	37	54	105	156	121	161	169	201
$\log_{10}$ #params	6.1	6.1	8.1	8.1	8.1	8.2	8.1	8.1	8.1	8.2	7.1	7.3	7.4	7.6	7.8	6.9	7.3	7.5	7.2
Top-1 error	41.9	41.8	31.0	30.1	28.4	27.6	29.6	28.5	26.6	25.8	30.2	26.7	23.9	22.6	21.7	25.4	24.0	22.8	22.4
Top-5 error	19.6	19.4	11.4	10.8	9.6	9.1	10.2	9.6	8.5	8.2	10.9	8.6	7.1	6.4	5.9	7.8	6.2	7.0	6.4

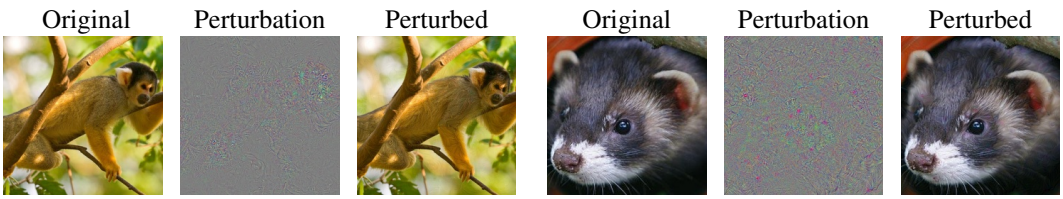


Figure 7: AIP for an ImageNet classifier. The perturbations are generated at $L_2 = 1 \times 10^{-4}$.

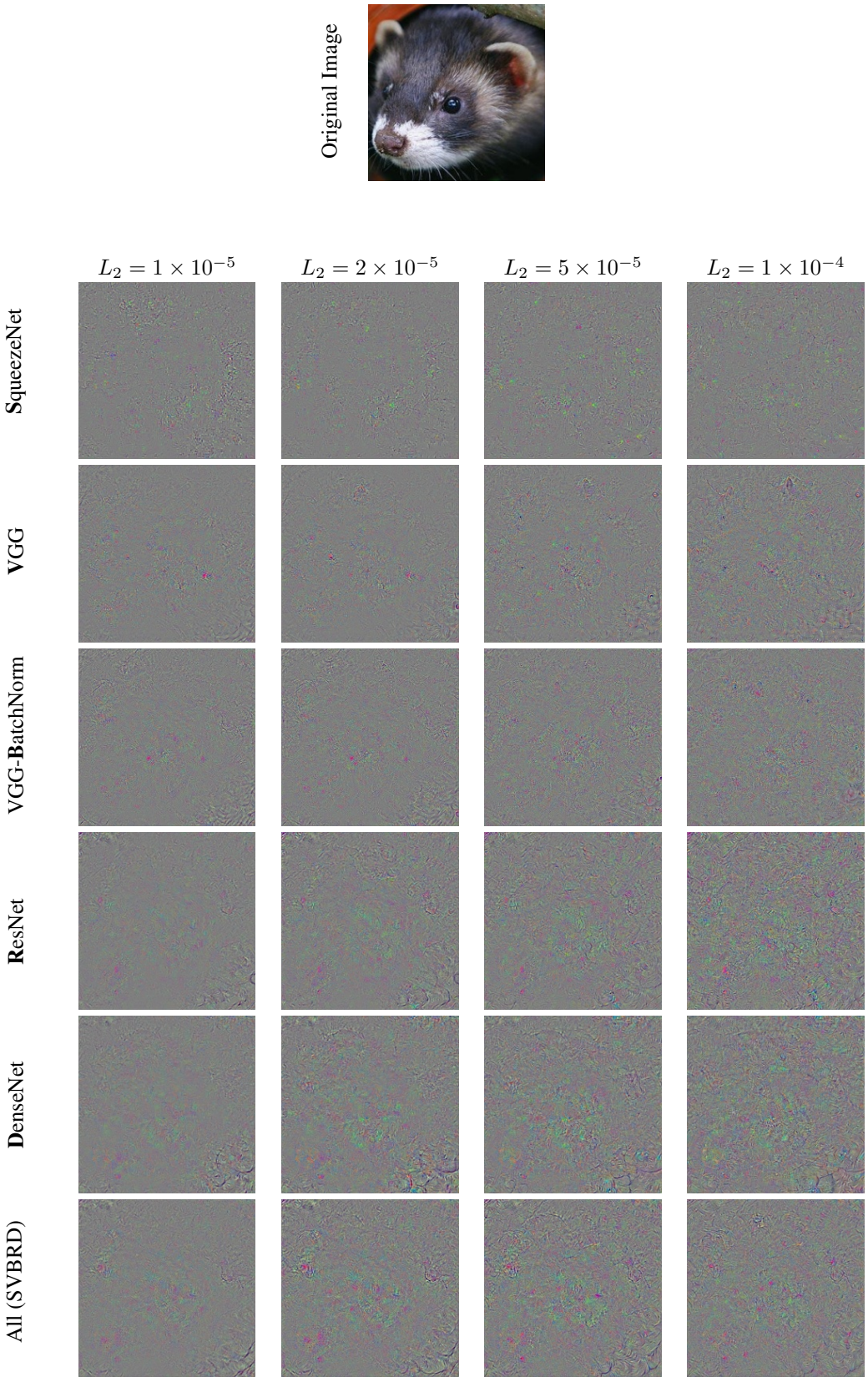


Figure 8: Adversarial perturbations for the same input image (top) generated with diverse ImageNet classifier families (S, V, B, R, D, SVBRD) at different norm constraints. The perturbation images are normalised at the maximal perturbation for visualisation. We observe diverse patterns across classifier families within the same L_2 ball.

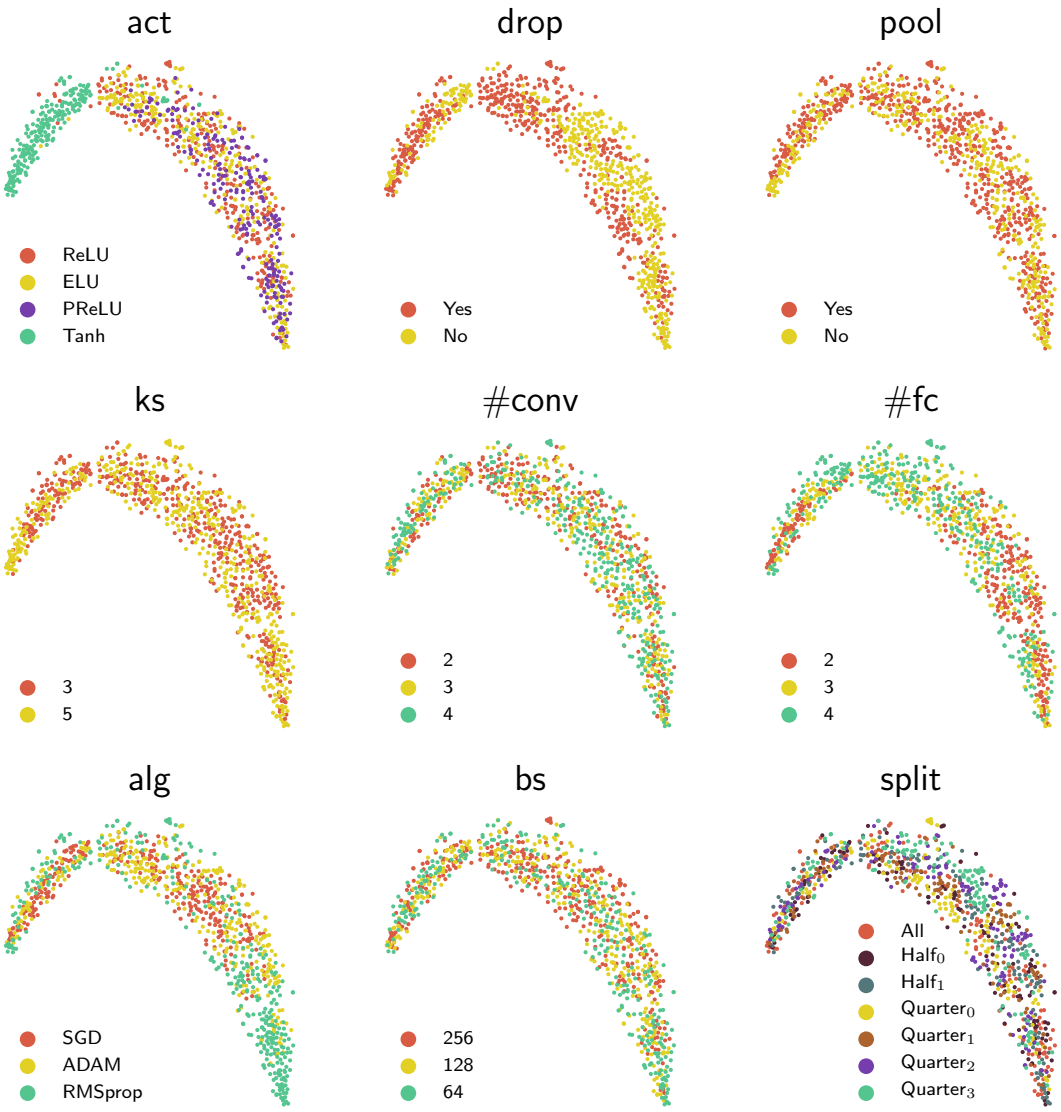


Figure 9: Probability query output embedded into 2-D plane via t-SNE. The same embedding is shown with different colour-coding for each attribute. These are the inputs to the `kennen-o` metamodel.

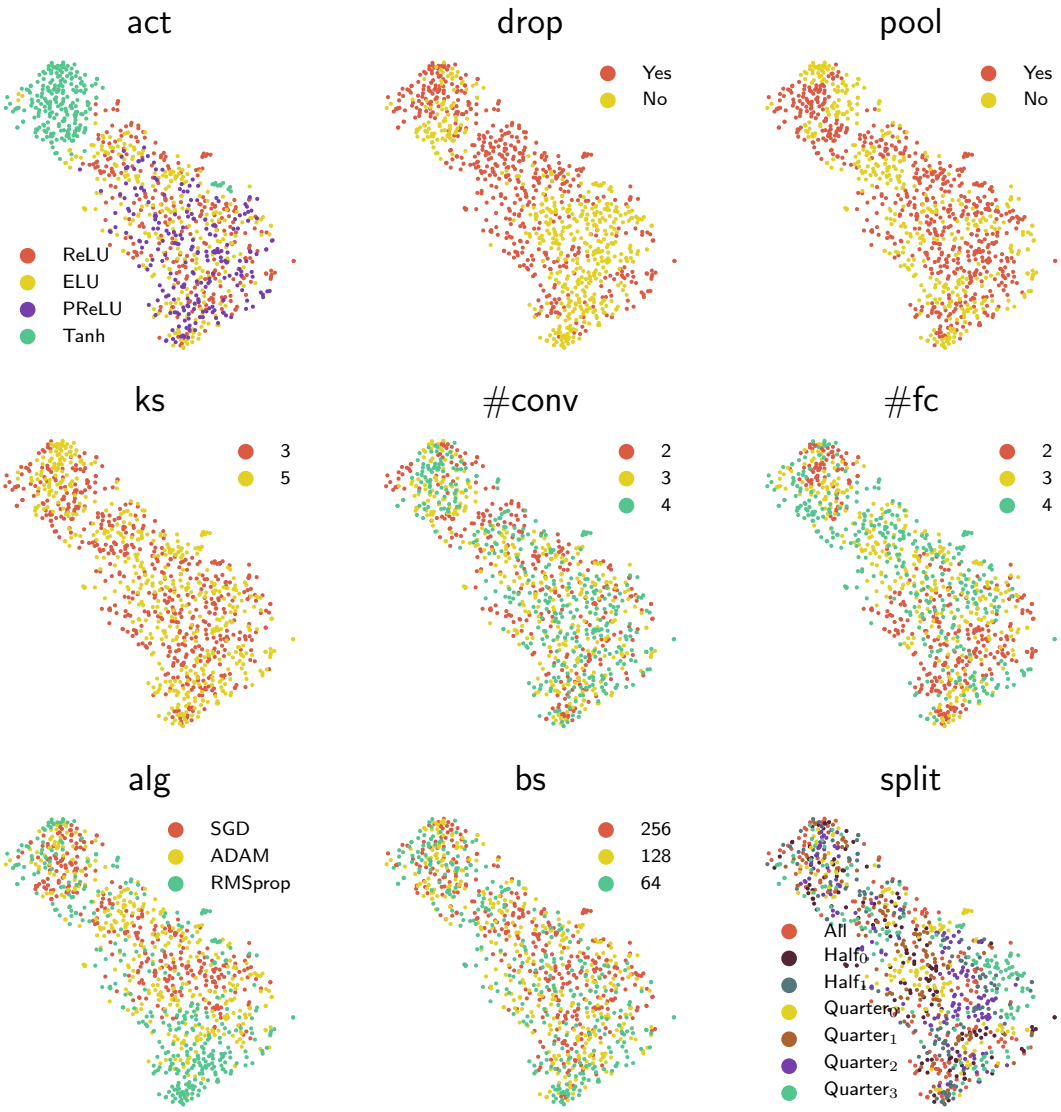


Figure 10: Probability query output embedded into 2-D plane via t-SNE. The same embedding is shown with different colour-coding for each attribute. These are the inputs to the `kennen-io` metamodel.

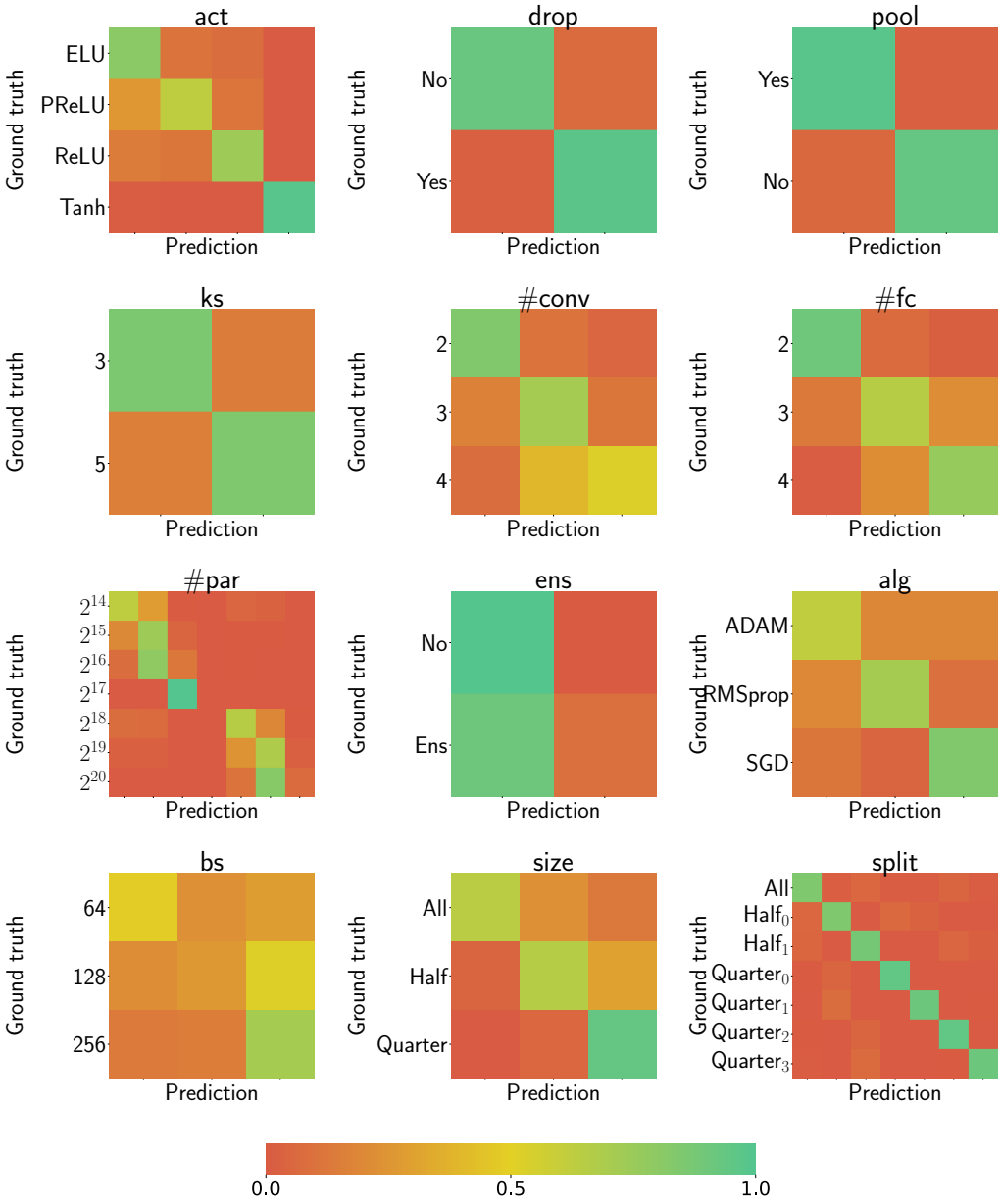


Figure 11: Confusion matrices for `kennen-o`.

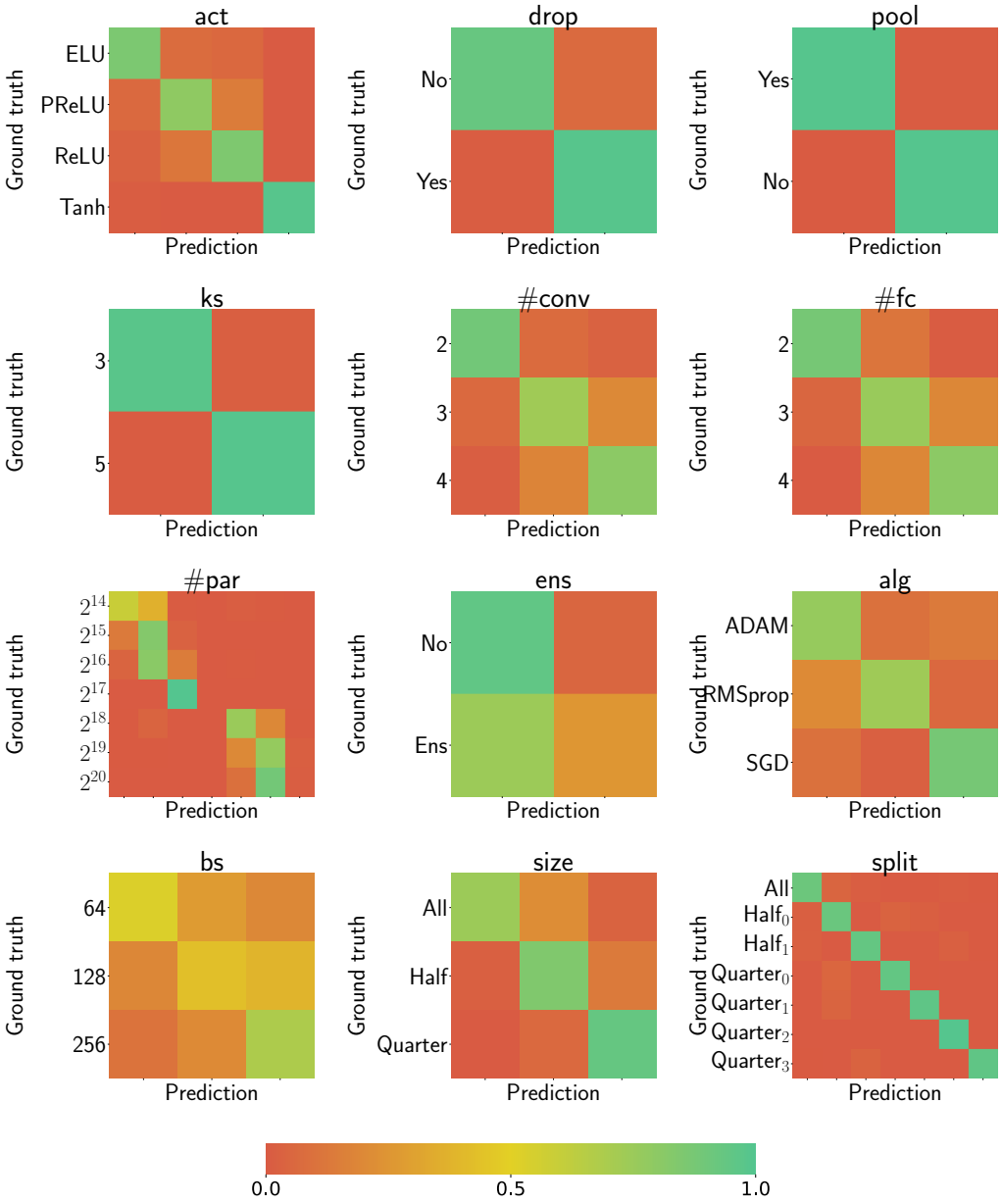


Figure 12: Confusion matrices for `kennen-io`.