Be nice to your neurons

Initialisation, Normalisation, Regularisation and Optimisation

Petar Veličković

Artificial Intelligence Group
Department of Computer Science and Technology, University of Cambridge, UK

COMPGI23—Introduction to Deep Learning, University College London **14 November 2017**

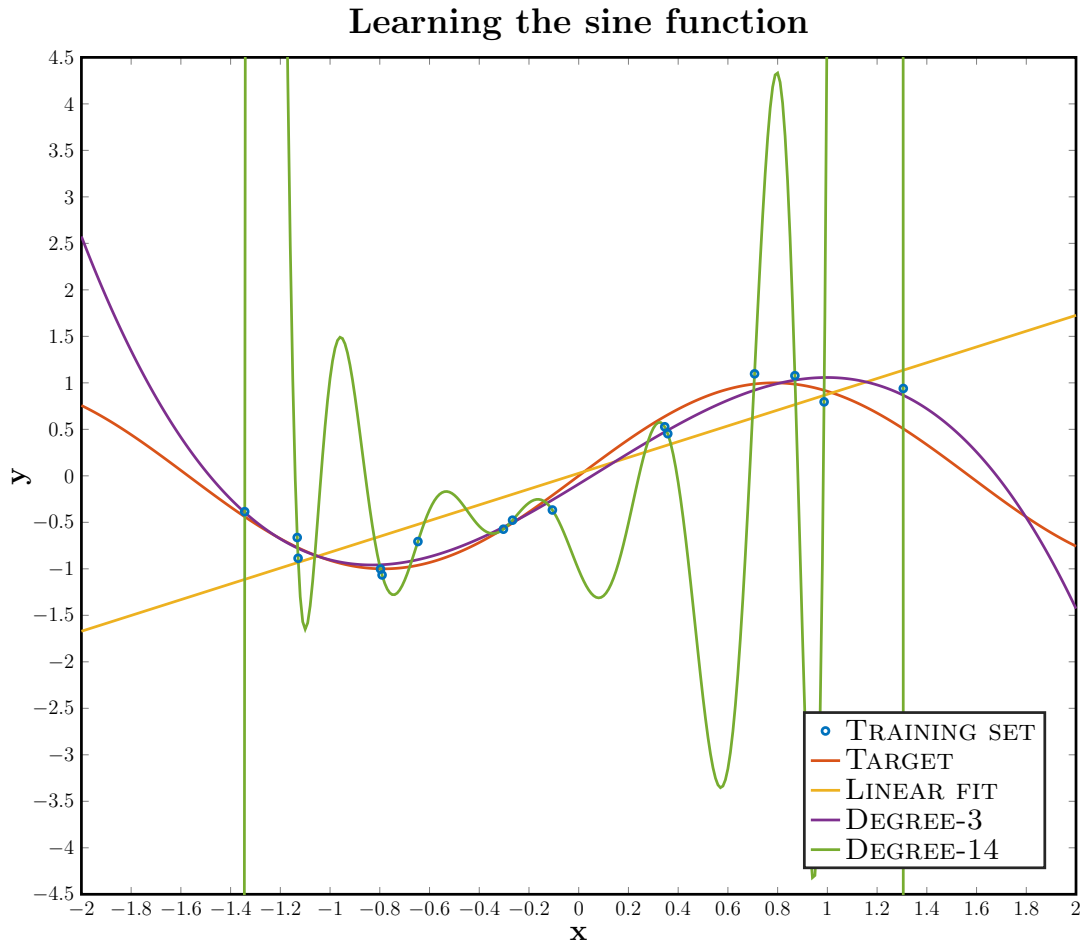
Introduction

- In this lecture, I will cover several essential techniques that can help *simplify neural network training*.
- These will primarily be techniques for making the network's actions more *robust* and *generalisable*, across every stage of its layer pipeline. We will cover simple, commonly-used methods as well as some ongoing bleeding-edge research.
- The title holds a *double meaning*: if you treat your networks well, you'll require less effort to properly optimise them! :)

A (rough) outline for today

1. **Regularisation**: Forcing our networks to find *good parameters* while simultaneously being *constrained* in some other way (hopefully, one that discourages parameter choices that overfit!)
2. **Initialisation**: Choosing *initial parameters* ("starting points" for training) for the network in order to encourage good behaviour in early stages of training.
3. **Normalisation**: Making sure that our network consistently receives *well-behaved signals* (in terms of magnitudes and statistics) at all stages of its processing pipeline.
4. **Optimisation**: Actually deciding how to use the computed gradients to *update the network parameters* in a stable and adaptive way.

Combatting **overfitting**



Last time I just said *"beware"*. . . now let's see what we can do.

Regularisation

- Overfitting can often be alleviated using **regularisation**.
- Broadly speaking, regularisation corresponds to making the model optimise the same problem, under *additional constraints* that somehow *restrict* the set of parameters available to the model during training.
- With proper regularisation, *many* deep learning models can learn to fit the provided training data in a controllable fashion, regardless of their nominal parameter count.

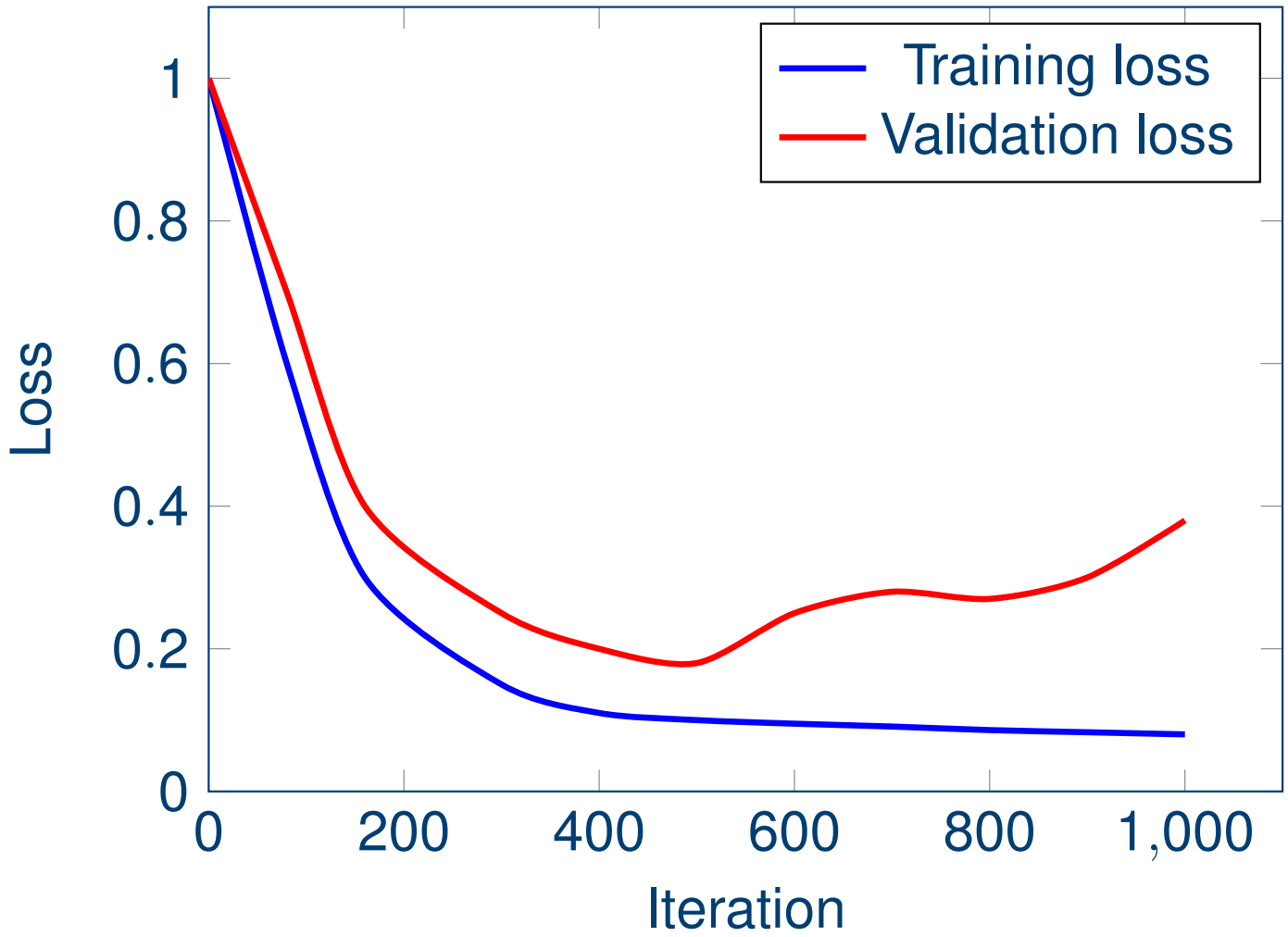
Early stopping

- Hyperparameters in a neural network are commonly optimised by using the obtained loss on a *validation set* as a proxy.
- For arguably the least intrusive form of regularisation, we can (re)use the validation set to learn the *number of training iterations* as a hyperparameter.
- This leads us to the technique of **early stopping**, which is now near-ubiquitously used across deep learning applications.

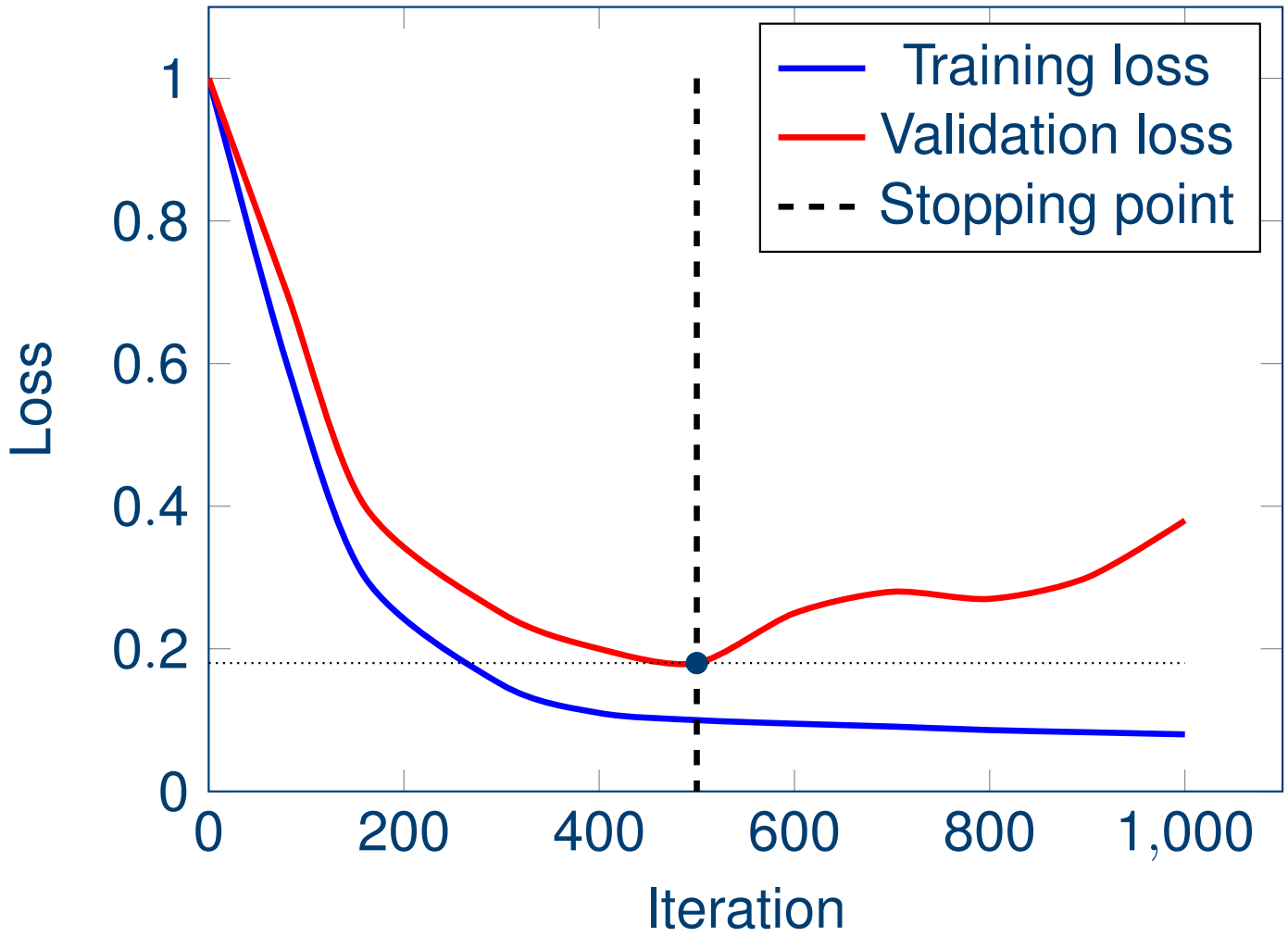
Early stopping: the essentials

- While training, keep track of the model parameters that achieved the *best validation loss* so far.
- Stop training when the loss hasn't improved for a set number of iterations (the **patience** hyperparameter).
- Restore the best-performing model (and potentially evaluate it on a testing dataset).

Early stopping in action



Early stopping in action



Early stopping: analysis

- We are basically assuming that validation performance will be a good surrogate for testing performance.
  - Very simple, and usually very effective (esp. if large valid. set)!
- Often desirable to use, as it does not require modifying anything else about the training or model setup itself!
- Downsides include the cost of storing a copy of the parameters (*often negligible*) and separating out a validation set (*but can retrain on it afterwards*).

Early stopping: regularisation

- How exactly does early stopping *regularise* the model?
- Assume, for simplicity:
  - Initial model parameters $\mathbf{w}$;
  - A fixed learning rate of η;
  - That the learning is performed over τ iterations;
  - Early stopping at the τ'th iteration;
  - That the maximal weight gradient is bounded by $|g_{max}|$.
- This means that, initially, each parameter is allowed to vary in the range $w_i \pm \eta\tau|g_{max}|$.
- However, an early stopped model's parameters can only vary in the range $w_i \pm \eta\tau'|g_{max}|$—this is a *regularising constraint*, limiting the set of parameters *reachable* from $\mathbf{w}$!

Early stopping in Keras

Simple to do using the `EarlyStopping` callback!

```
model.fit(X_train, Y_train, batch_size=batch_size,
epochs=num_epochs, verbose=1, validation_split=0.1,
callbacks=[EarlyStopping(monitor='val_loss', patience=5)])
```

L_2 regularisation

- Now, we turn our attention to a regulariser that *directly restricts* the desirable values of weights.
- We do this by further constraining our loss function, $\mathcal{L}$ (e.g. cross-entropy or squared error), by adding a penalty term, which penalises weights that deviate too much from zero:

$$\tilde{\mathcal{L}}(\vec{x}, \vec{y}, \vec{w}) = \mathcal{L}(\vec{x}, \vec{y}, \vec{w}) + \frac{\lambda}{2}\|\vec{w}\|^2$$

  where λ is a hyperparameter.
- We have arrived at the L_2 *regularisation* technique (also called **weight decay**), as we seek to restrict the L_2 norm of $\vec{w}$.

Implication on training dynamics

- Recall the gradient descent update rule:

$$\vec{w} \leftarrow \vec{w} - \eta \frac{\partial \mathcal{L}}{\partial \vec{w}}$$

- The gradient of the added L_2 penalty is $\lambda \vec{w}$. Therefore:

$$\vec{w} \leftarrow \vec{w} - \eta \left(\frac{\partial \mathcal{L}}{\partial \vec{w}} + \lambda \vec{w} \right)$$

$$\vec{w} \leftarrow (1 - \eta\lambda)\vec{w} - \eta \frac{\partial \mathcal{L}}{\partial \vec{w}}$$

- This forces the weight vector to *shrink* in magnitude at each step, prior to applying the gradient update!

L_2 regularisation: notes

- Nowadays, L_2 regularisation has been somewhat abandoned in favour of more recent ideas—but is still often very useful.
- Properly choosing λ is important!
  - Too low: negligible effects;
  - Too high: weights set to $\vec{0}$.
- **N.B.** The regularisation effect would still be present if we tended to a point other than the origin, i.e. if we optimised

$$\tilde{\mathcal{L}}(\vec{x}, \vec{y}, \vec{w}) = \mathcal{L}(\vec{x}, \vec{y}, \vec{w}) + \frac{\lambda}{2} \| \vec{w} - \vec{w}' \|^2$$

  but this is rarely done in practice (why?).

L_2 regularisation in Keras

Simply attach a `kernel_regularizer` to the model!
(**N.B.** we often do not regularise the bias, as it is less overfit-prone)

```
y = Dense(64, activation='relu',
kernel_regularizer=regularizers.l2(0.0001))(x)
```

Aside: L_1 regularisation

- While the L_2 norm might have seemed like a natural choice to optimise, nothing stops us from using a different (L_p) norm.
- Especially, L_1 regularisation is another popular approach. In this scenario we penalise the L_1 norm:

$$\tilde{\mathcal{L}}(\vec{x}, \vec{y}, \vec{w}) = \mathcal{L}(\vec{x}, \vec{y}, \vec{w}) + \lambda \sum_i |w_i|$$

  The gradient update is no longer as simple to analyse—but it can be shown (details omitted) that this will force several weights to be (near-)zero, **sparsifying** the model.
- This makes L_1 regularisation useful for, among other things, *feature selection* (cf. LASSO).

Ensemble methods

- While a final trained model may not perfectly generalise, we may find that the exact nature of the faults it makes will depend on the (*stochastic*) progress of the training procedure.
- This means that we could be able to extract better performance if we *train several models* and *average their predictions*!
- This technique is known as **ensembling**.

An overview of the potential of ensembling

- Assume, for simplicity, that we are optimising a squared error loss, and that model i makes error ϵ_i (with variance $\mathbb{E}(\epsilon_i^2) = v$ and covariance $\mathbb{E}(\epsilon_i \epsilon_j) = c$).
- The error made by an average of k such models is $\frac{1}{k}\sum_{i=1}^{k} \epsilon_i$. The expected squared error is then:

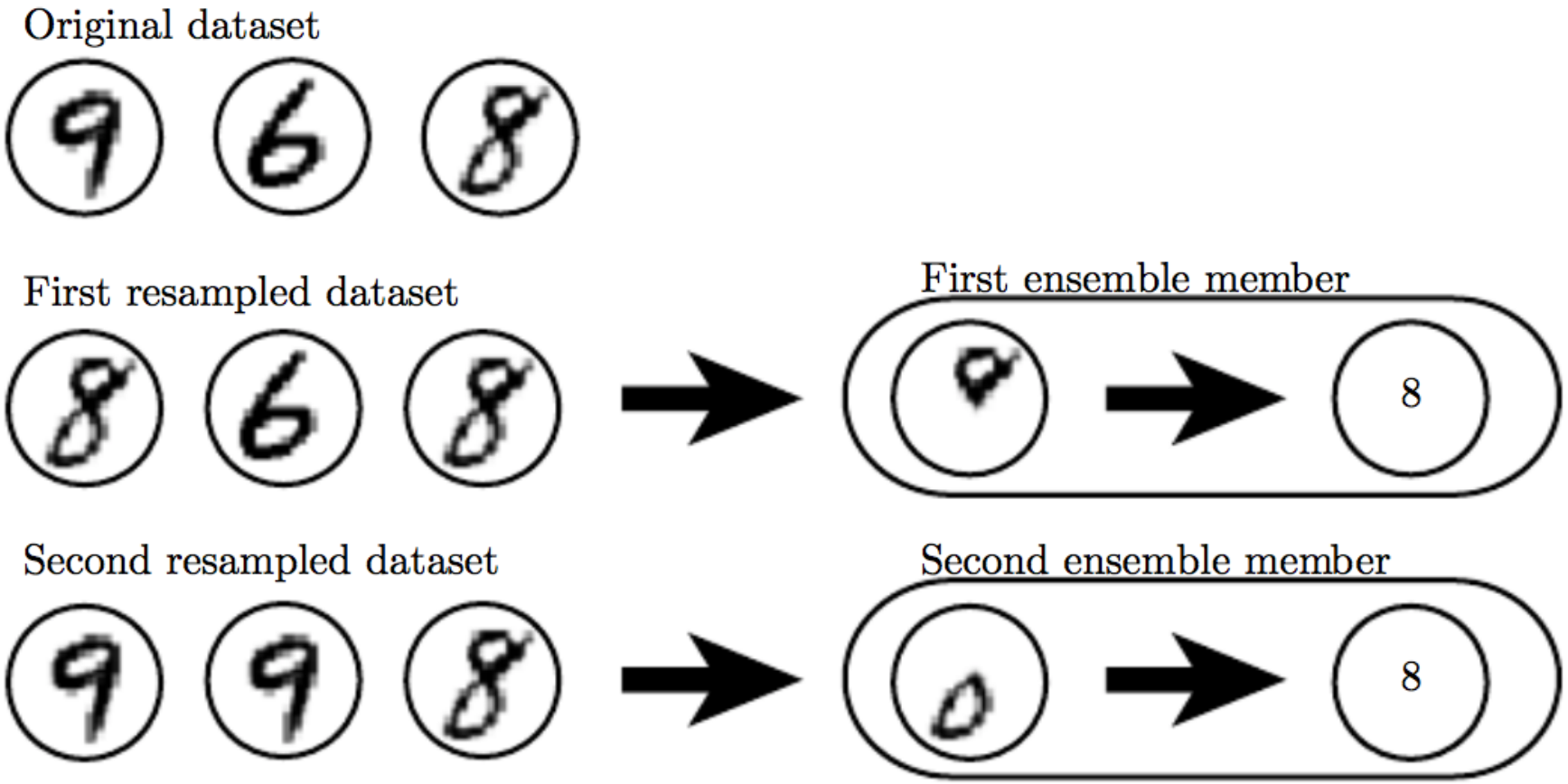
$$\mathbb{E}\left(\left(\frac{1}{k}\sum_{i=1}^{k}\epsilon_i\right)^2\right) = \frac{1}{k^2}\mathbb{E}\left(\sum_{i=1}^{k}\left(\epsilon_i^2 + \sum_{j\neq i}\epsilon_i\epsilon_j\right)\right) = \frac{1}{k}v + \frac{k-1}{k}c$$

- If errors are not fully correlated ($c < v$), this causes an improved generalisation performance!

Bagging

- If we want to decorrelate our errors while ensembling a model with the same hyperparameters (desirable!), a good technique to use is **bootstrap aggregating** (*bagging*).
- Here we expose each copy of the model to a *different dataset*, by sampling k examples from the training set with replacement.
- This means that, if the dataset size is unchanged ($k = n$), approximately two-thirds of the examples will be present in the resampled dataset (with the remainder replaced by duplicates).

Bagging in action (*Goodfellow et al., 2016*)



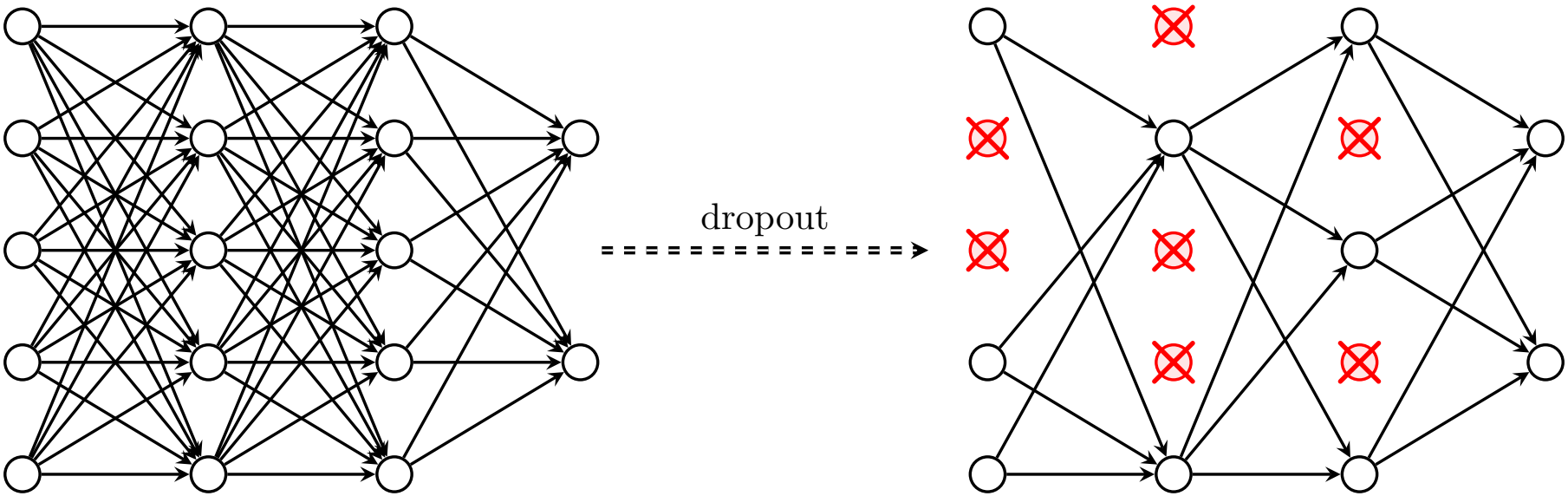
Notes on using ensembles

- Ensembling is a remarkably powerful technique—often *required* for winning machine learning competitions.
- However, it should usually be avoided when benchmarking machine learning algorithms (for scientific publications), especially when comparing against single-model approaches.
- Lastly, note that not all ensembling methods are regularisers—e.g. a technique called **boosting** will create models with *increased* capacity.

Reliance on neurons

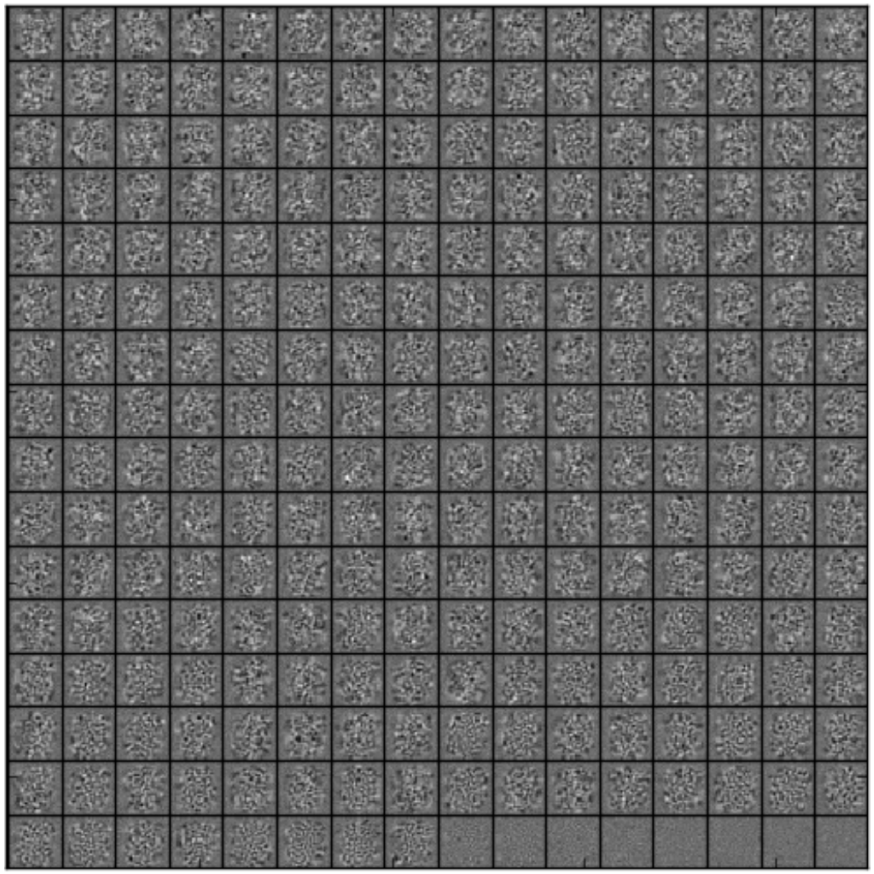
- When trained without additional constraints, neural networks will tend to encourage *highly specialised* neurons.
- Even worse, there might be neurons that perform *poorly*, with several other neurons there just to *correct for its mistakes*.
- Both of these make the network overly *reliant* on actions of specific neurons, which generalises poorly (if only one neuron fails, it might compromise many others!).
- $\Longrightarrow$ could be useful to force the network not to rely on the existence of a neuron...

Dropout (*Srivastava et al., 2014*)

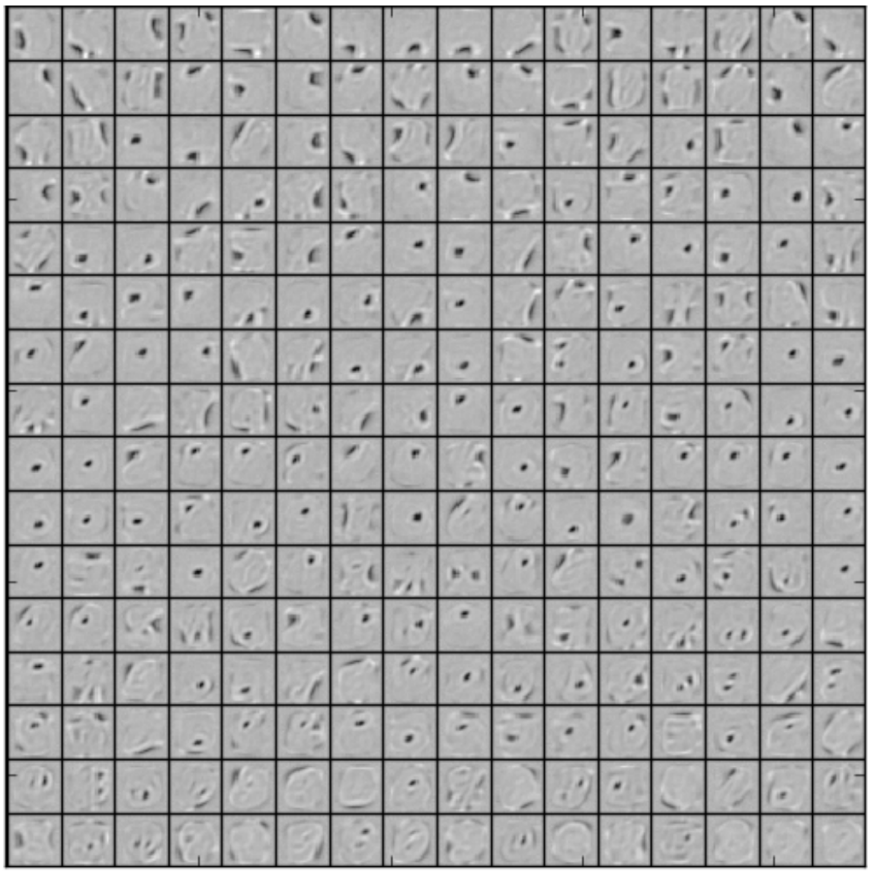


- Randomly "kill" each neuron in a layer with probability p, during training only. Also scale the output of remaining neurons by $\frac{1}{1-p}$, to preserve expected value.

Dropout in action (*Srivastava et al., 2014*)



(a) Without dropout

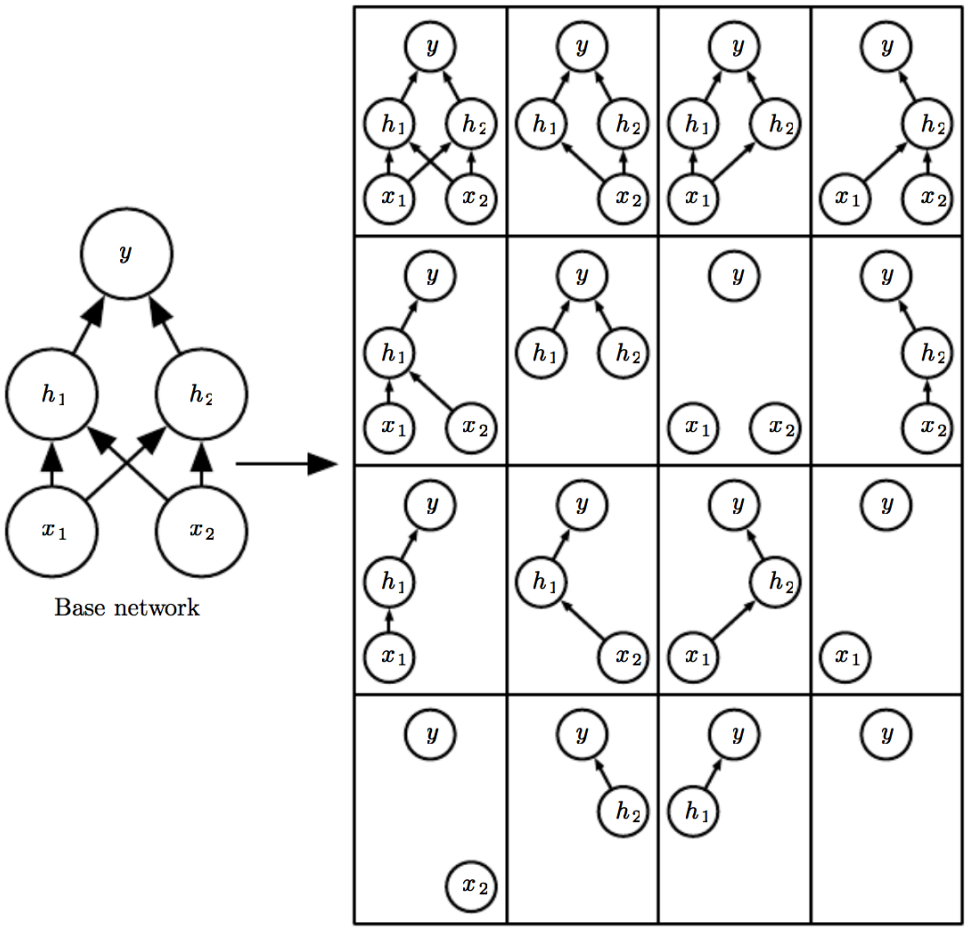


(b) Dropout with $p = 0.5$.

An alternate look at dropout

- When using dropout, at each iteration a different *model* is used (depending on which hidden units are dropped), by sampling mask elements from a *Bernoulli*(p) distribution.
- This means that dropout can also be seen as a computationally cheap way of *creating a large ensemble of neural networks* (with 2^n possibilities for n neurons).
- Although there is no formal proof for the fact that running the full model at evaluation time preserves the required ensemble properties, this technique performs really good in practice!

The ensembled networks (*Goodfellow et al., 2016*)



Notes on dropout

- Dropout is a very desirable method, given its high effectiveness as a regulariser and favourable computational cost.
- Typically, good values of p lie in $0 \leq p \leq 0.5$, but it is sometimes possible to require even higher values.
- Rough rule-of-thumb:
  - Fully-connected layers are the most prone to overfitting, and applying dropout with $p = 0.5$ to their output is often appropriate.
  - Overfitting is not as common with convolutional layers, but a weaker dropout ($p \approx 0.25$) applied there could still be helpful.
  - Occasionally, it becomes useful to dropout inputs to the network.
- As dropout is turned off for validation/testing, note that the training loss may not initially be better than validation loss!

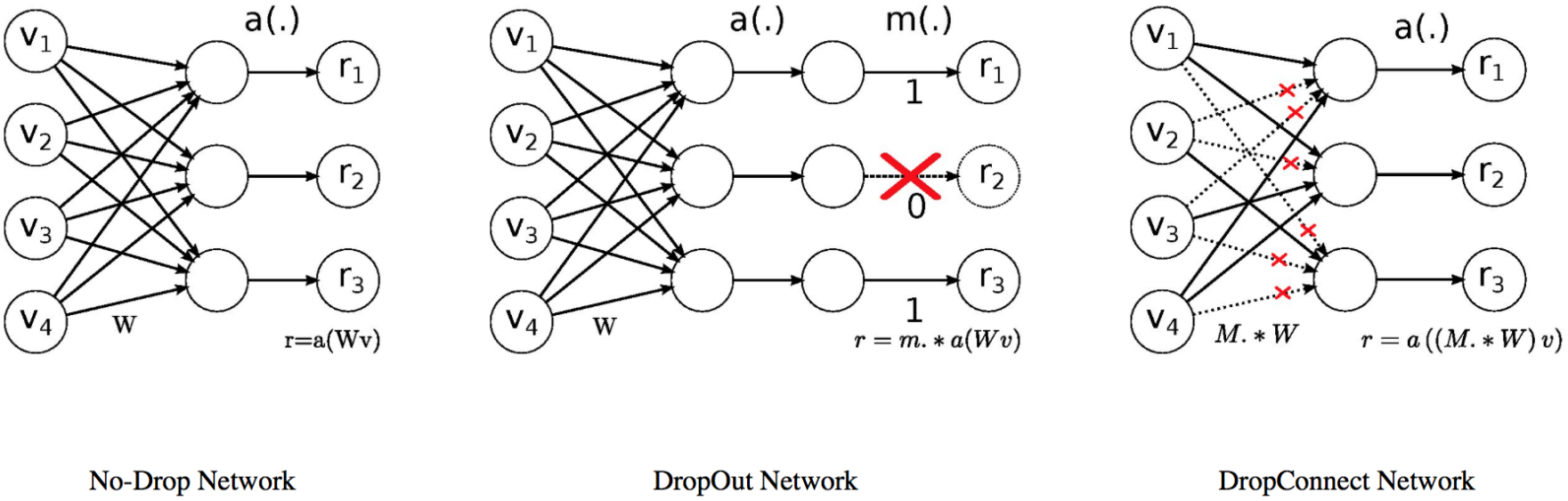
Dropout in Keras

Extremely simple—just apply the `Dropout` layer!

```
x_drop = Dropout(0.5)(x)
```

Aside: DropConnect (*Wan et al., 2013*)

- Generalises dropout by dropping *weights* rather than *outputs*.



No-Drop Network DropOut Network DropConnect Network

- Harder to do both training (as it requires a different mask for each example) and inference (requiring Gaussian moment matching). However, some impressive results (MNIST SOTA)!

Concrete dropout (*Gal et al., 2017*)

- It is possible to relax the $z \sim Bernoulli(p)$ distribution to a continuous version (the *Concrete distribution*):

$$\tilde{z} = \sigma\left(\frac{1}{t} \cdot (\log p - \log(1-p) + \log u - \log(1-u))\right)$$

  where p is the dropout probability, $u \sim U(0,1)$, t is a temperature parameter, and σ is the logistic sigmoid function.

- This is now **differentiable** by p—can learn the hyperparameter directly by *gradient descent*!

- The technique is known as **Concrete dropout**.

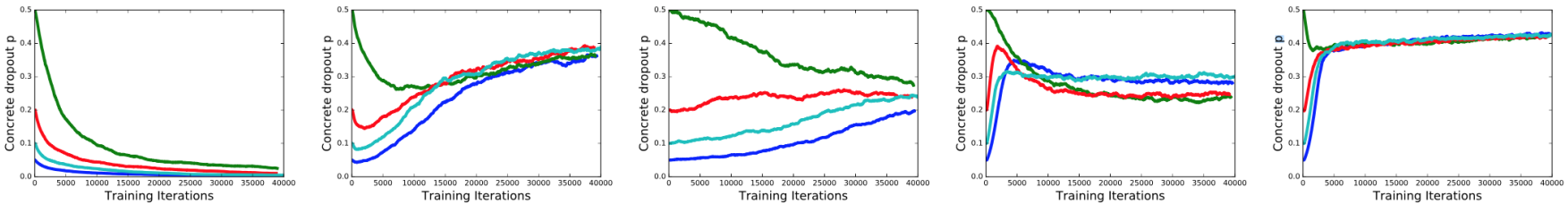
Concrete dropout (*Gal et al., 2017*)

- The authors provide a `ConcreteDropout` wrapper for Keras, which allows one to trivially apply it to any layer's output:
  `y = ConcreteDropout(Dense(32, activation='relu'))(x)`

- The reference implementation may be found at:
  `https://github.com/yaringal/ConcreteDropout`

Concrete dropout (*Gal et al., 2017*)

- Directly learning *p* from training data (requiring no validation dataset) makes this technique very attractive for *small-data problems* as well as *reinforcement learning*!
- The learnt values of the concrete dropout's *p* do not always coincide with manual tuning results! (e.g. for an encoder-decoder network for image segmentation:)



(a) $L = 0$ (b) $L = 1$ (c) $L = n/2$ (d) $L = n - 1$ (e) $L = n$

Figure 8: Learned Concrete dropout probabilities for the first, second, middle and last two layers in a semantic segmentation model. p converges to the same minima for a range of initialisations from $p = [0.05, 0.5]$.

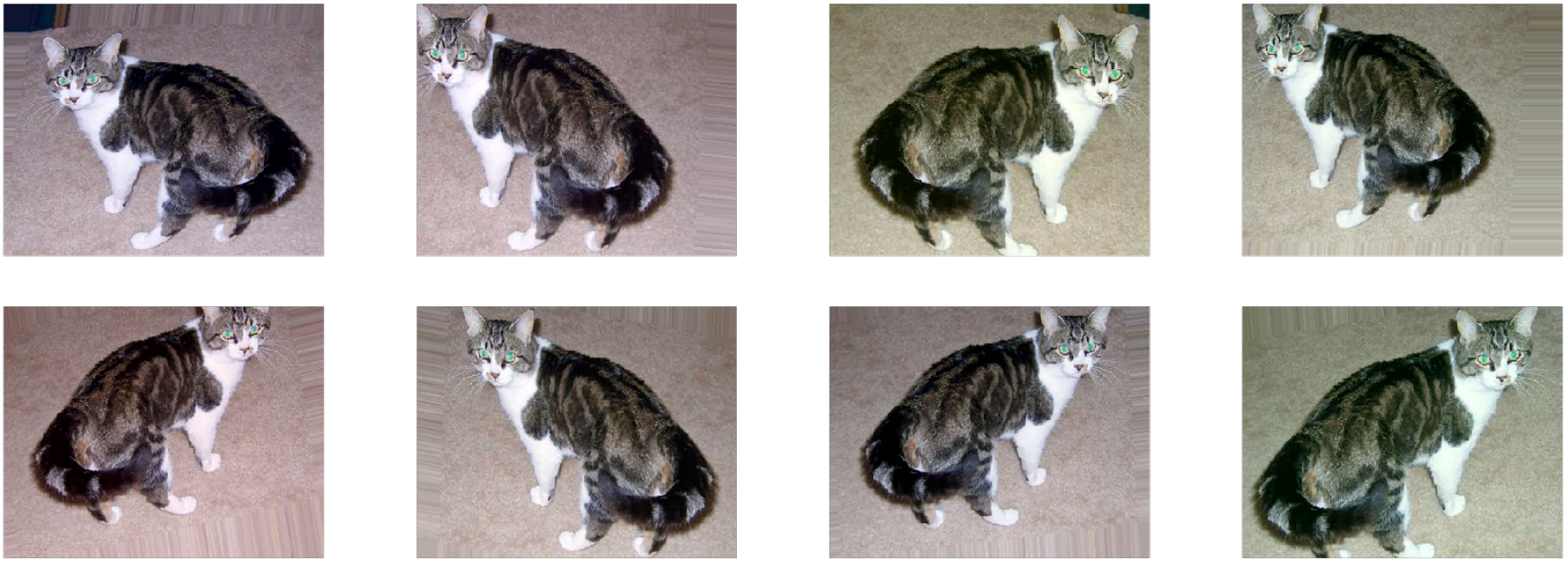
Data augmentation

- As mentioned in my Week 2 lecture, the best way to improve a network's generalisation performance is to *gather more training data*—but this is not always feasible!
- A simple manner of gathering more data is *randomly transforming the training inputs* in a way that makes *predictable changes* to the ground-truth outputs.
  - Most commonly, for classification problems we can choose transformations that do not change the class.
- We arrive at the common technique of **data augmentation**.

Data augmentation in computer vision

- This technique is best visualised (and most commonly applied) on an **image classification** dataset:
  - Applying random shifts/scales/rotations/crops/. . . to the input image will usually not change the object class of the image!
  - *Careful* with some transformations (such as horizontal flips).
- Now the network will *never see the same example twice during training*—significantly lowering the potential for overfitting.
- Furthermore, it will be encouraged to learn to be resistant to such perturbations in the input!

Data augmentation in action



(Image) Data augmentation in Keras

Make advantage of the `ImageDataGenerator` class!

```
datagen = ImageDataGenerator(width_shift_range=0.1,
height_shift_range=0.1, rotation_range=10)
```

~~`model.fit(...)`~~

```
model.fit_generator(datagen.flow(
X_train, y_train, batch_size=32),
steps_per_epoch=len(X_train) / 32, epochs=100)
```

Being nice to our signals

- We will now focus on techniques that are primarily designed to make our data representations *more convenient to work with* as the data passes through the network.

- As such, they're not designed with regularisation in mind—but many of them end up being useful regularisers nonetheless!

Input normalisation

- To avoid numerical issues with signals of overly large magnitude or variance, it is almost-always helpful to *normalise* the input of the network to have more controlled statistics.
- Typically, this can be done by computing the sample means, $\vec{\mu}$, and sample standard deviations, $\vec{\sigma}$, across each feature within the training set, and transforming the input data, x, by

$$\tilde{x}_i = \frac{x_i - \mu_i}{\sigma_i}$$

- Note that we **cannot observe the test data**, so the same values of $\vec{\mu}$ and $\vec{\sigma}$ are used for evaluating!
- *Many of the techniques to follow will assume that we've normalised our inputs!*

Network initialisation

- We now revisit a problem explored in Week 2—appropriately *initialising network weights*.
- We've shown how *unsupervised pre-training* can be a good initialiser (albeit costly in terms of the added training time).
- Since then, such methods have largely been replaced with drawing initial weights from *carefully crafted probability distributions*.
  - These often perform equivalently or better to unsupervised pre-training, for only a fraction of the effort.

Initialisation in Keras

- We will usually draw each weight from the **uniform** or **normal** distribution centered at zero, i.e. $U(-x, x)$ or $\mathcal{N}(0, \sigma)$.
  - Thus far, no significant difference in the two has been found.
  - We only need to specify the **standard deviation**, σ!
- Usually initialise biases to zero.
- Keras will allow us to specify the initialiser using the `kernel_initializer` parameter:

```
y = Dense(64, activation='relu',
kernel_initializer='ones')(x)
```

LeCun initialisation (*LeCun et al., 1998*)

- An early development towards such initialisers is the *LeCun initialisation*. Consider what happens to the variance of a linear neuron's output, assuming its weights w_i and inputs x_i are *uncorrelated* and *zero-mean*:

$$Var\left(\sum_{i=1}^{n_{in}} w_i x_i\right) = \sum_{i=1}^{n_{in}} Var(w_i x_i)$$
$$= \sum_{i=1}^{n_{in}} Var(W)\,Var(X) = n_{in}\,Var(W)\,Var(X)$$

  where n_{in} is the *fan-in*—the number of inputs to the neuron.
- This approximation is appropriate for *sigmoid* activations as well (as they're roughly linear in their unsaturated region).

LeCun initialisation (*LeCun et al., 1998*)

- Typically, we will want to choose our weights to *preserve the variance* of the input signal as it passes through the network.
- This implies that we should set

$$Var(W) = \frac{1}{n_{in}}$$

- Use `'lecun_uniform'` or `'lecun_normal'` in Keras.
- To make it work nicely for *deeper networks*, we will need another constraint...

Xavier initialisation (*Glorot and Bengio, 2010*)

- Signals do not just go *forward* in a network—they are also *propagated backwards* when computing gradients!
- During this procedure, a neuron will accumulate gradients from the neurons it directly sends output to, scaled by the relevant weights. Let n_{out} be the *fan-out*—the number of such neurons.
- A similar argument as before now mandates we should set

$$Var(W) = \frac{1}{n_{out}}$$

in order to preserve the variance of the gradient update signals.

Xavier initialisation (*Glorot and Bengio, 2010*)

- Typically, we cannot satisfy both constraints simultaneously. *Xavier initialisation* instead aims for their average:

$$Var(W) = \frac{2}{n_{in} + n_{out}}$$

- Now a *ubiquitously* used initialisation scheme.

- Use `'glorot_uniform'` (**default!**) or `'glorot_normal'` in Keras.

He initialisation (*He et al., 2015*)

- We've thus far assumed a *linear* layer. This was somewhat okay for sigmoid activations, but *not* for ReLU (which is most commonly used)!
- *He initialisation* rectifies the analysis to take into account the behaviour of ReLUs (details omitted), obtaining:

$$Var(W) = \frac{2}{n_{in}}$$

- Enabled **surpassing human performance** on ImageNet!
- Use `'he_uniform'` or `'he_normal'` in Keras.

Orthogonal initialisation (*Saxe et al., 2014*)

- An alternate approach to initialising is to *not initialise weights independently* from one another—this may be helpful in obtaining some desirable properties.
- The best-known example of this is *orthogonal initialisation*, which constrains the initial weight matrix $\mathbf{W}$ to be **orthogonal**, i.e. $\mathbf{W}^T\mathbf{W} = \mathbf{I}$.
- This has two desirable properties:
  - The transformation is **norm-preserving**, i.e. $\|\mathbf{W}\vec{x}\| = \|\vec{x}\|$—this helps *combat vanishing/exploding gradients* early on.
  - Its columns are **orthonormal**, i.e. $\vec{w}_i^T \vec{w}_j = \delta_{ij}$—this encourages the network to learn *varied input features*.
- Use `'orthogonal'` in Keras.

LSUV (*Mishkin and Matas, 2015*)

- Perhaps the best approach to initialisation is one that is **data-driven**—*tuned to the features of the input data*. One interesting development in this direction is *Layer-sequential unit variance (LSUV)* initialisation.

- Initialise layer-by-layer, starting with an orthogonally-initialised weight matrix, then tuning on minibatches of training data until the variance of the layer's output signals becomes close to 1.

LSUV (*Mishkin and Matas, 2015*)

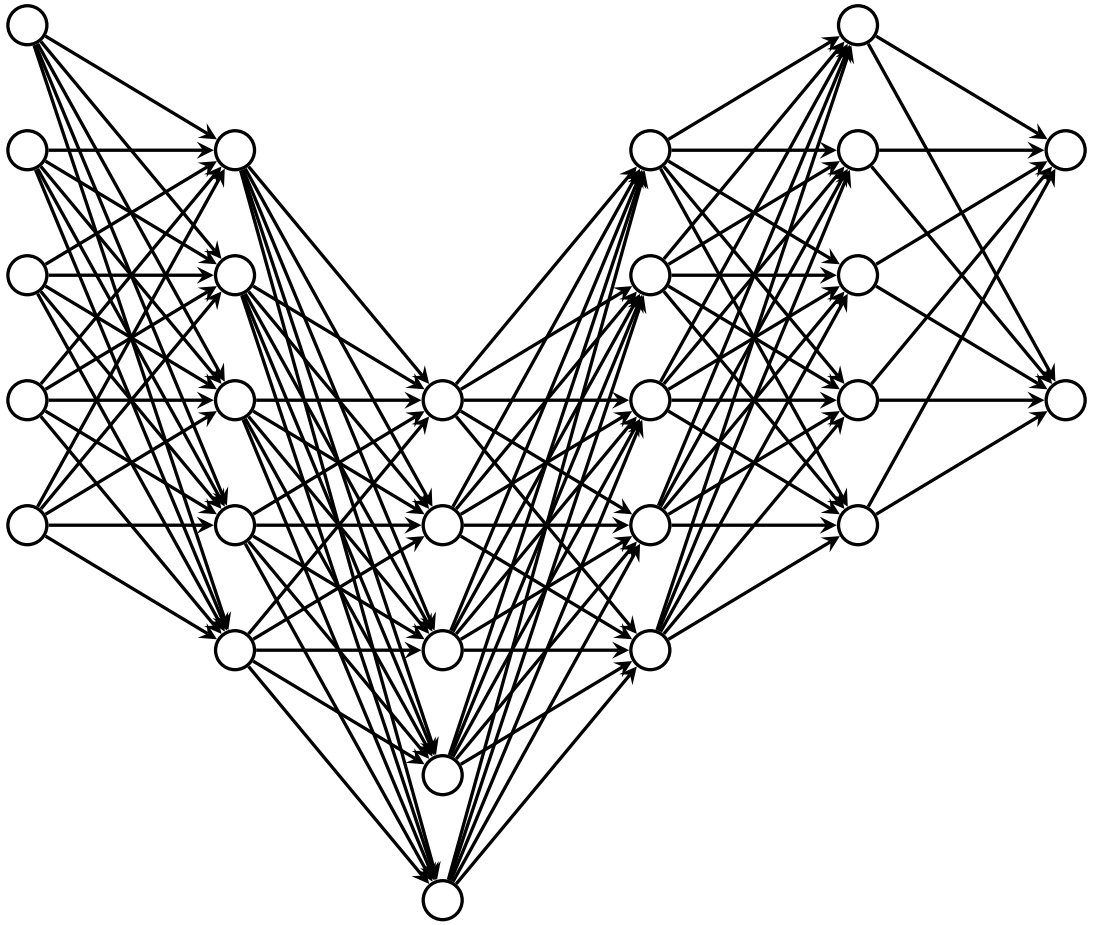
Algorithm 1 Layer-sequential unit-variance orthogonal initialization. L – convolution or full-connected layer, W_L - its weights, B_L - its output blob., Tol_{var} - variance tolerance, T_i – current trial, T_{max} – max number of trials.

Pre-initialize network with orthonormal matrices as in Saxe et al. (2014)
for each layer L **do**
  while $|Var(B_L) - 1.0| \geq Tol_{var}$ and $(T_i < T_{max})$ **do**
    do Forward pass with a mini-batch
    calculate $Var(B_L)$
    $W_L = W_L \,/\, \sqrt{Var(B_L)}$
  end while
end for

Internal covariate shift

- Thus far, we've focussed our attention at making the input signal well-behaved, and, *initially*, making it propagate through the network nicely in both directions.
- However, as training progresses, especially for deeper networks, it is to be expected that weights are pushed in a direction that will change the intermediate layer statistics—an effect known as the **internal covariate shift**.
- This internal covariate shift is an important problem in deep networks, since any neuron deeper in the network needs not only to *do its job properly*, but also to **adapt to inputs that are changing over time**.

Internal covariate shift, visualised



The solution is quite simple—**renormalise** periodically!

Batch normalisation (*Ioffe and Szegedy, 2015*)

- The first—and still by far most widely used—approach to reducing internal covariate shift is renormalising the signal across each training minibatch (**batch normalisation**).
- Let the outputs of the current layer across the current batch be $\mathcal{B} = \{x_1, \ldots, x_m\}$. Then:

$$\mu_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i \quad \sigma^2_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2$$

$$\hat{x}_i = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \varepsilon}} \qquad y_i = \gamma\hat{x}_i + \beta$$

  where γ and β are **trainable** (allowing the network to *revert*)!
- Now ubiquitously used for training deep architectures.

Notes on batch normalisation

- At test time, use the entire training set's statistics to fix parameters of μ and σ (usually as a *moving average*):

$$\mu \leftarrow \mu + \alpha(\mu_{\mathcal{B}} - \mu) \qquad \sigma \leftarrow \sigma + \alpha(\sigma_{\mathcal{B}} - \sigma)$$

- *Regularises* the network! Similarly to data augmentation, the signals the network will see will rarely repeat (as signals for one example now depend on its entire minibatch)!
- Sometimes applied to the **inputs** of the network directly as a substitute for preprocessing!
- Simple to include in a Keras model, by leveraging the `BatchNormalization` layer:

```
y = BatchNormalization()(x)
```

Limitation of batch normalisation

- One issue of batch normalisation is its dependence on minibatch statistics.
- This makes the technique harder to apply in, e.g.:
  - *online learning*, where batch size is forced to be 1;
  - working with *variable-length inputs* and RNNs;
  - *noise-sensitive* applications such as reinforcement learning.
- We will now survey a few recently proposed techniques for overcoming this issue.
- There does not seem to be a clear consensus on which technique is best to use in these circumstances—attempting several of them is the best approach for a new problem.

Layer normalisation (*Ba et al., 2016*)

- Under the intuition that changes in output of one layer will tend to cause highly correlated changes in the summed inputs for the next layer, *layer normalisation* proposes normalising all signals across the layer, for a single training example:

$$\mu_l = \frac{1}{H}\sum_{i=1}^{H} a_i \qquad \sigma_l^2 = \frac{1}{H}\sum_{i=1}^{H}(a_i - \mu_l)^2$$

  where H is the number of neurons in the layer, and a_i is the output of the i-th neuron.

- This is followed by a learnable scale and shift, as in batch normalisation. **N.B.** each example is normalised independently!

Weight normalisation (*Salimans and Kingma, 2016*)

- *Weight normalisation* proposes to speed up the optimisation procedure of neural networks by making the norm of their weights *explicitly trainable*.
- This is achieved through the following reparametrisation:

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

  where g and $\vec{v}$ are *trainable* by gradient descent.
- A data-driven methodology is also suggested to initialise $g = \frac{1}{\sqrt{Var\left(\frac{\vec{v}\cdot\vec{x}}{\|\vec{v}\|}\right)}}$, after fixing an initial $\vec{v}$, across a single training minibatch of layer inputs $\vec{x}$.

Batch renormalisation (*Ioffe, 2017*)

- Proposes an extension to break the discrepancy of training and inference transformations of batchnorm.
- As before, compute μ and σ as moving averages of $\mu_{\mathcal{B}}$ and $\sigma_{\mathcal{B}}$, and use those for inference.
- During training, introduce a *correcting scale and shift*, which compensates for the discrepancies of μ and σ to $\mu_{\mathcal{B}}$ and $\sigma_{\mathcal{B}}$:

$$\frac{x_i - \mu}{\sigma} = \frac{x_i - \mu_{\mathcal{B}}}{\sigma_{\mathcal{B}}} \cdot r + d$$

This implies that $r = \frac{\sigma_{\mathcal{B}}}{\sigma}, d = \frac{\mu_{\mathcal{B}} - \mu}{\sigma}$. In practice, the values of r and d are clipped (initially fixing them to 1 and 0, respectively, and progressively widening the range of allowed values).

Residual connections (*He et al., 2015*)

- Training very deep networks has been very problematic for many years. Batch normalisation is only one of the many techniques that have helped significantly simplify the process in recent years.
- Another remarkable technique are *residual (skip) connections*, which enabled a 152-layer network to win the ImageNet competition in 2015.
- Key observation: stacking too many layers limits not only validation, but also **training** performance!
- But, if the added layers are useless, the network could just be encouraged to make them compute the **identity** function!

Residual connections (*He et al., 2015*)

- Essentially, allows the input to additively **shortcut** to a latter stage of the network's pipeline.

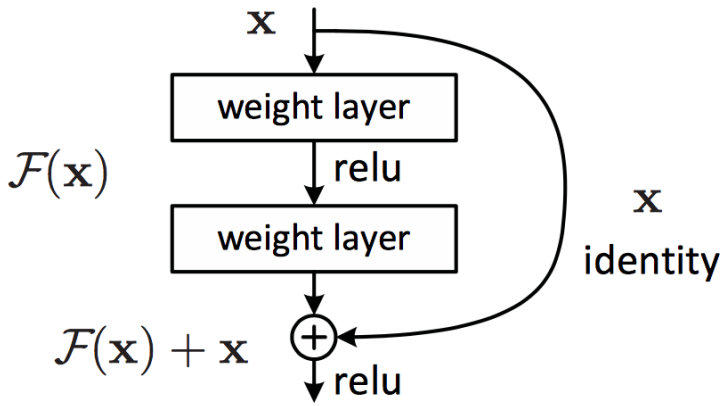
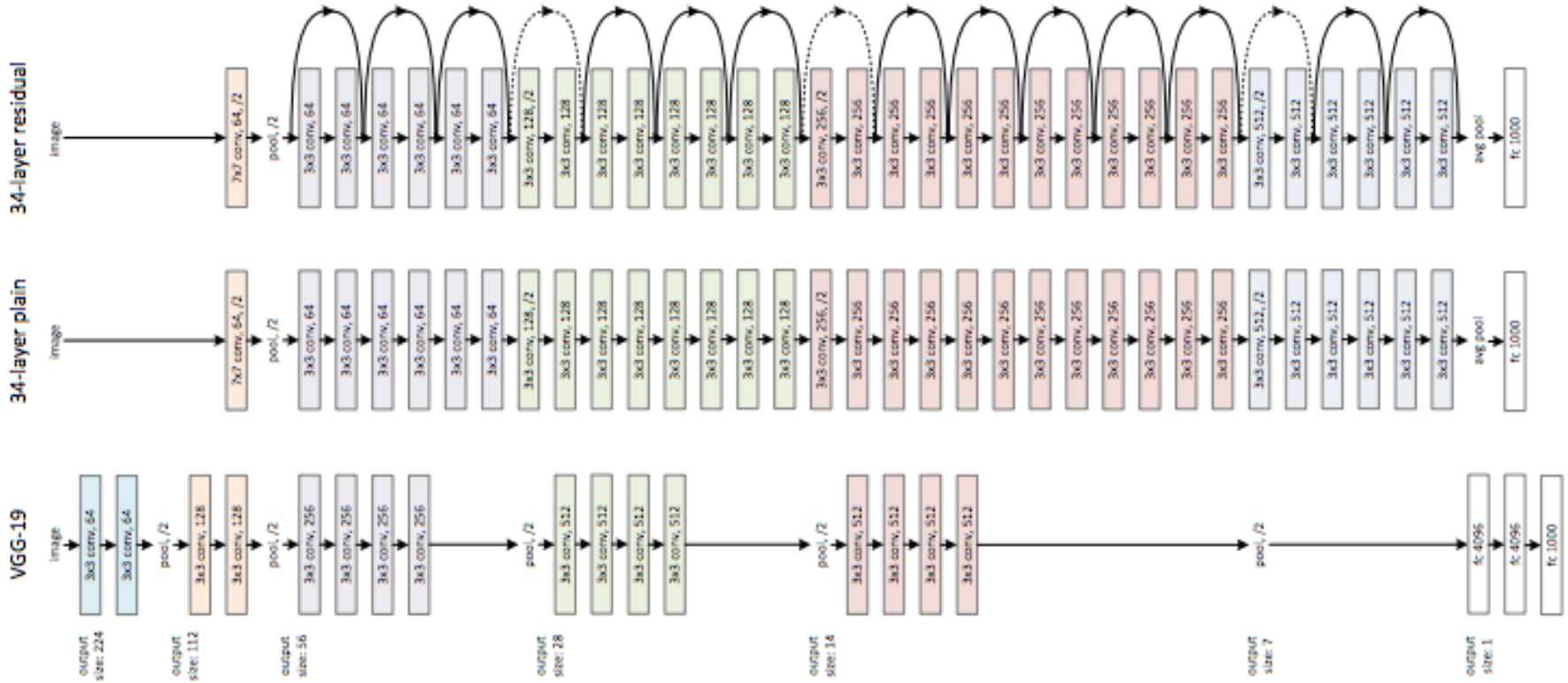


Figure 2. Residual learning: a building block.

- Lets the network effectively **choose its own depth**. . . a very powerful and versatile concept!

Residual networks (*He et al., 2015*)



`https://github.com/fchollet/keras/blob/master/keras/applications/resnet50.py`

Advanced ReLU activations: PReLU

- To conclude this section, we will cover two kinds of activation functions that are more general versions of the ReLU function, and have seen considerable usage in recent years.

- The parametric ReLU (**PReLU**), proposed by He *et al.*, allows for some (learnable) amount of output signal for negative input values:

$$PReLU(x_i) = \begin{cases} x_i & x_i > 0 \\ a_i x_i & x_i \leq 0 \end{cases}$$

  Here, a_i are learnable parameters—one per each feature.

Advanced ReLU activations: ELU

- Another substantial development in the field of activation functions is the exponential linear unit (ELU), proposed by Clevert *et al.*
- The ELU extends the ReLU into the negative values:

$$ELU(x) = \begin{cases} x & x > 0 \\ \alpha(\exp(x) - 1) & x \leq 0 \end{cases}$$

  Here, α is a hyperparameter; often set to 1.
- Critically, unlike the PReLU, here the output of the function **saturates** for too low values—keeping the negative signals under control.

Advanced ReLU activations (*Clevert et al., 2015*)

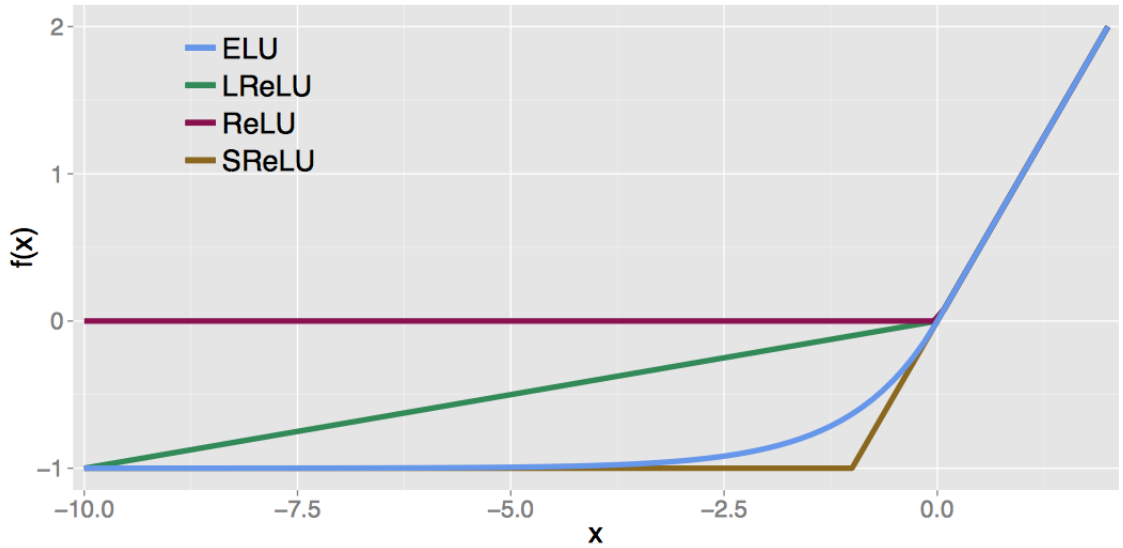


Figure 1: The rectified linear unit (ReLU), the leaky ReLU (LReLU, $\alpha = 0.1$), the shifted ReLUs (SReLUs), and the exponential linear unit (ELU, $\alpha = 1.0$).

Gradient descent optimisers

- We turn our attention to the *learning algorithm* employed to train our neural networks.
- This area has seen significant improvement over the past years, and has now completely simplified the training of many common neural network models (often in conjunction with the other regularising techniques already enumerated here).
- I will, for the most part, follow the brilliant overview by Sebastian Ruder, which I highly recommend: `http://ruder.io/optimizing-gradient-descent/`

Vanilla minibatch SGD

- Recall the standard minibatch stochastic gradient descent framework:

$$\vec{w} \leftarrow \vec{w} - \eta \frac{\partial \mathcal{L}_{\mathcal{B}}(\vec{w})}{\partial \vec{w}}$$

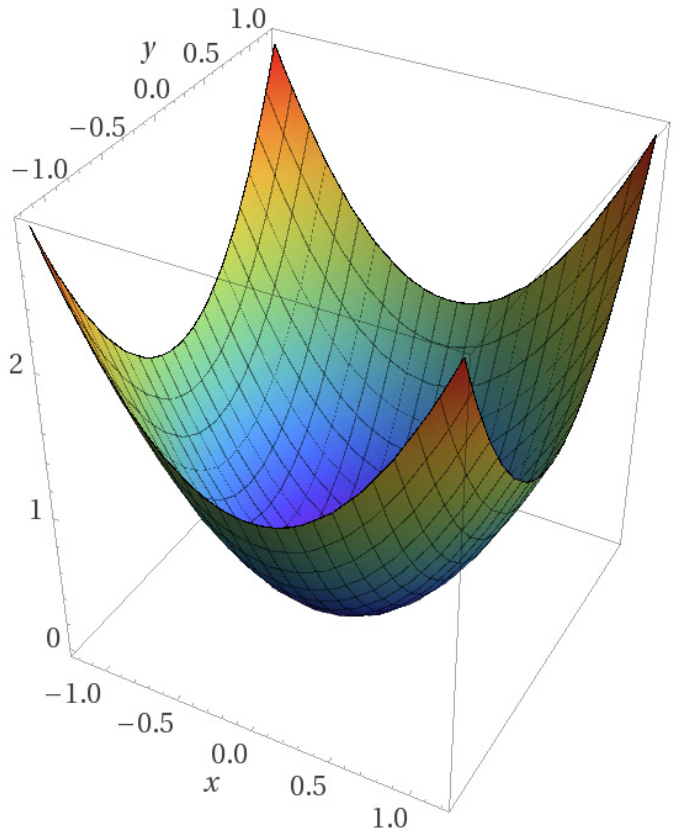
  where $\mathcal{L}_{\mathcal{B}}$ is the loss on a sampled minibatch $\mathcal{B}$ of training examples, and η the learning rate.

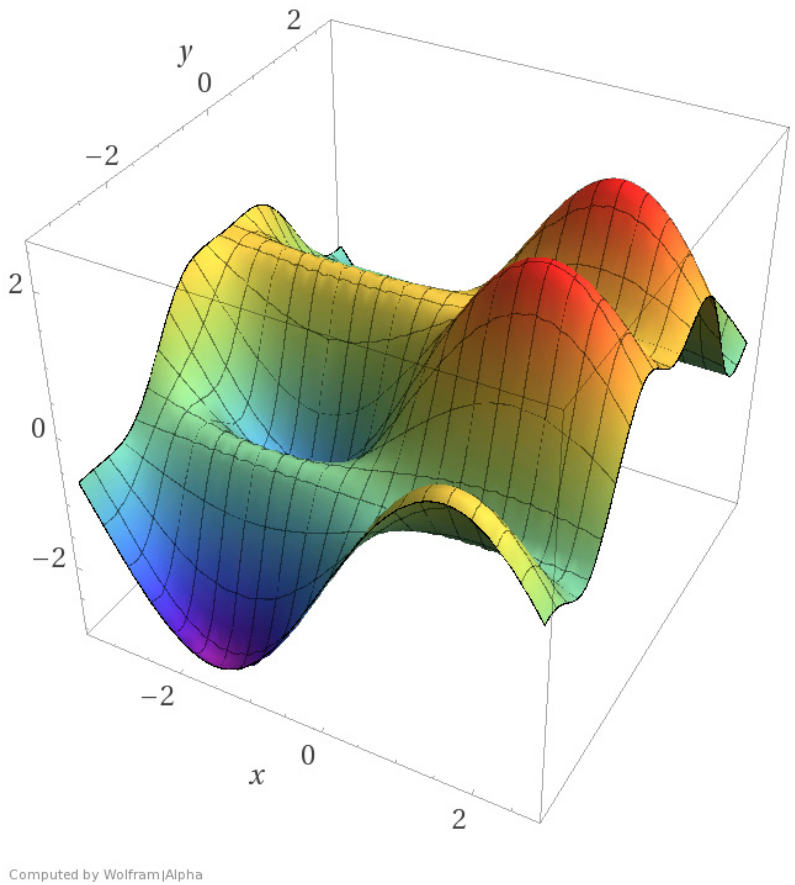
- **N.B.** SGD and backpropagation are different concepts! Backpropagation is used to efficiently compute the gradients $\frac{\partial \mathcal{L}_{\mathcal{B}}(\vec{w})}{\partial \vec{w}}$, whereas SGD is used to update the weights, *assuming the gradients are already computed*.

Limitation of vanilla SGD

- Choosing a proper value of η can be notoriously difficult—with poorly chosen values either significantly slowing down convergence or hindering it.
- One way to address this issue is *learning rate annealing*, where η starts with a larger value and is gradually reduced—but this schedule needs to be set-up upfront, and therefore is **unable to adapt** to the characteristics of the loss function's surface.
- Furthermore, the same learning rate is used for updating all the weights, which may be inappropriate (different weights may require different scales)!

Recall: getting trapped in *saddle points*





Momentum (*Qian, 1999*)

- One very simple approach to improving the performance of SGD is *momentum*—take a step in an accumulated direction:

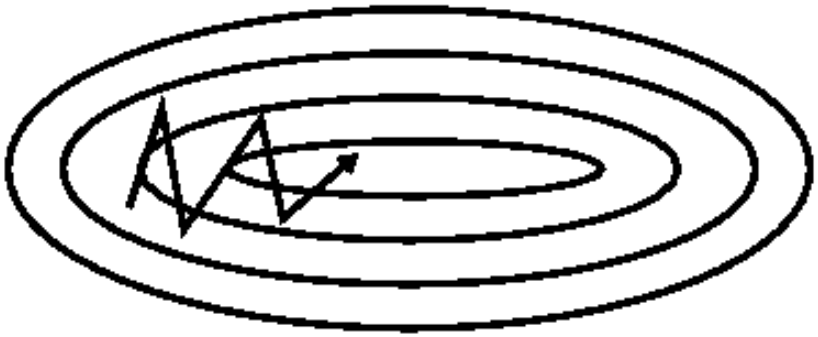
$$\vec{v}_t = \gamma \vec{v}_{t-1} + \eta \frac{\partial \mathcal{L}_{\mathcal{B}}(\vec{w})}{\partial \vec{w}} \qquad \vec{w} \leftarrow \vec{w} - \vec{v}_t$$

  where γ is a hyperparameter, often set to a value near 0.9.

- This has the effect of *amplifying* gradient updates in the directions that are *consistent* across timesteps, and *reducing* them in *oscillating* directions—speeding up convergence and improving stability.

Effects of momentum

Without momentum



With momentum

Vanilla SGD + momentum is still widely used in research, as it provides a lot of control (at the expense of having to put in more effort in optimising the learning rate schedule).

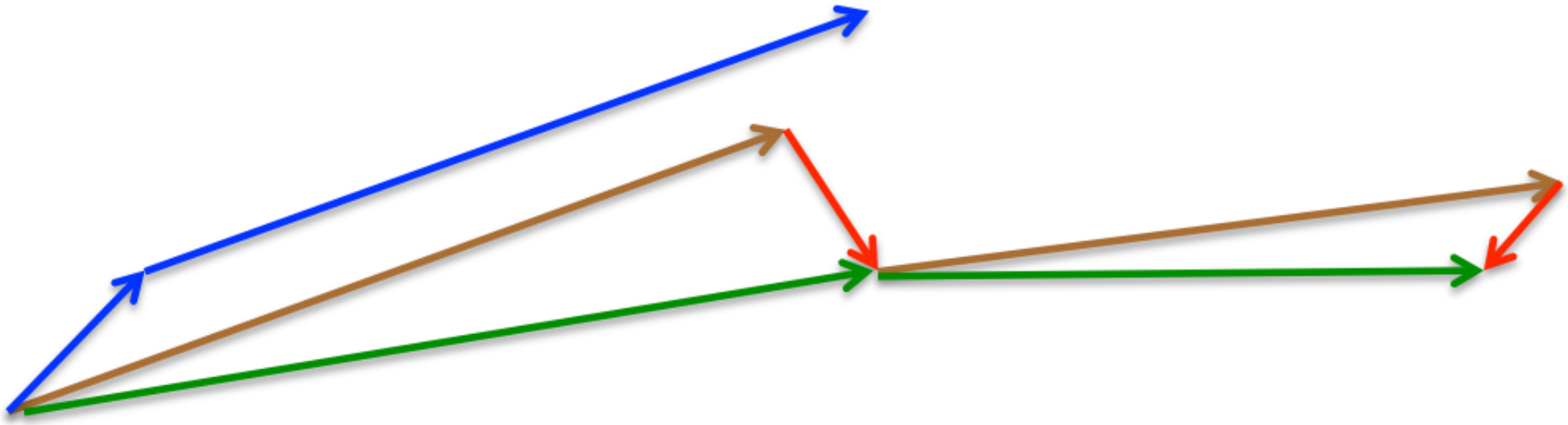
Nesterov accelerated gradient (*Nesterov, 1983*)

- Nesterov's **NAG** method allows us to further improve the "look-ahead" capability of momentum.
- Since we're going to apply the $\gamma \vec{v}_{t-1}$ update no matter what, we might as well use this position to evaluate the gradient on:

$$\vec{v}_t = \gamma \vec{v}_{t-1} + \eta \frac{\partial \mathcal{L}_{\mathcal{B}}(\vec{w} - \gamma \vec{v}_{t-1})}{\partial \vec{w}} \qquad \vec{w} \leftarrow \vec{w} - \vec{v}_t$$

- This prevents us from making too eager jumps, and increases *responsiveness*.

Effects of NAG



Standard momentum: **blue** vector—compute the local gradient, then take a big step in the direction of accumulated gradient.

Nesterov momentum: **green** vector—first take the step in the direction of the accumulated gradient (**brown** vector), then correct based on the local gradient there (**red** vector).

Momentum methods in Keras

- All the methods described so far can be deployed to a Keras model by modifying the model's compilation:

```
model.compile(loss='categorical_crossentropy',
optimizer=SGD(lr=0.01, momentum=0.9, nesterov=True))
```

Adagrad (*Duchi et al., 2011*)

- First widely used method that **adapts** the learning rate for each parameter *individually*.
- Let $\vec{g}_t$ be a vector that holds the gradient at time t, i.e. $g_{t,i} = \frac{\partial \mathcal{L}(\vec{w})}{w_i}$. Then AdaGrad modifies SGD by adapting the learning rate for w_i using historical gradient values:

$$w_{t+1,i} \leftarrow w_{t,i} - \frac{\eta}{\sqrt{\mathbf{G}_{t,ii} + \varepsilon}} \cdot g_{t,i}$$

where $\mathbf{G}_t$ is a diagonal matrix where $\mathbf{G}_{t,ii}$ is the *sum of squares of gradients* wrt w_i until timestep t.

Adagrad: notes

- To use in Keras, set `optimizer='adagrad'`.
- Intuition: weights that receive strong gradients less frequently should receive a stronger learning rate to compensate.
- Benefit: no longer need to explicitly worry about the value of η; often, setting it to a value like 0.01 is fine.
- Main weakness: values of **G** can only grow—learning rate eventually **vanishes**!

Adadelta (*Zeiler, 2012*)

- Designed to address the aggressive learning rate decay of Adagrad, *Adadelta* smooths the sum of squares of gradients with *exponential averaging*:

$$E(\vec{g}^2)_t = \gamma E(\vec{g}^2)_{t-1} + (1-\gamma)\vec{g}_t^2$$

  where γ is once again chosen to be around 0.9.

- Replacing $\mathbf{G}_t$ with $E(g^2)_t$, we arrive at the following parameter update (denoting the root mean square metric by *RMS*):

$$\Delta \vec{w}_t = -\frac{\eta}{\sqrt{E(\vec{g}^2)_t + \varepsilon}}\vec{g}_t = -\frac{\eta}{RMS(\vec{g})_t}\vec{g}_t$$

Adadelta (*Zeiler, 2012*), *cont'd*

- The authors note that the parameters in the update rule do not have the same units, so another exponential average (this time over the parameter updates) is defined:

$$E(\Delta \vec{w}^2)_t = \gamma E(\Delta \vec{w}^2)_{t-1} + (1-\gamma)\Delta \vec{w}_t^2$$

- Replacing the learning rate with the RMS of these updates up to time $t-1$ (as the update at time t is not immediately known) yields the Adadelta update rule:

$$\Delta \vec{w}_t = -\frac{RMS(\Delta \vec{w})_{t-1}}{RMS(\vec{g})_t}\vec{g}_t \qquad \vec{w}_{t+1} \leftarrow \vec{w}_t + \Delta \vec{w}_t$$

- To use in Keras, set `optimizer='adadelta'`.

RMSprop (*Tieleman and Hinton, 2012*)

- A method developed concurrently with Adadelta, with the same aim of rectifying the issues of Adagrad. Presented for the first time on Geoffrey Hinton's Coursera course on Neural networks!
- In fact, equivalent to the first update rule of Adadelta!

$$\vec{w}_{t+1} \leftarrow \vec{w}_t - \frac{\eta}{\sqrt{E(g^2)_t + \varepsilon}} \vec{g}_t$$

- To use in Keras, set `optimizer='rmsprop'`.

Adam (*Kingma and Ba, 2014*)

- Adaptive moment estimation (**Adam**) is currently the most popular adaptive optimisation algorithm.
- Computes *two* exponential averages: one of the *gradients* as well as the *squared gradients* (as Adadelta and RMSprop did):

$$\vec{m}_t = \beta_1 \vec{m}_{t-1} + (1 - \beta_1)\vec{g}_t \qquad \vec{v}_t = \beta_2 \vec{v}_{t-1} + (1 - \beta_2)\vec{g}_t^2$$

  where β_1 and β_2 are hyperparameters (with default values of 0.9 and 0.999, respectively).
- These tend to be biased towards zero early on, so the following *bias-corrected values* are recorded:

$$\hat{\vec{m}}_t = \frac{\vec{m}_t}{1 - \beta_1^t} \qquad \hat{\vec{v}}_t = \frac{\vec{v}_t}{1 - \beta_2^t}$$

Adam (*Kingma and Ba, 2014*), *cont'd*

- Now, use $\vec{\hat{m}}_t$ as the gradient direction (similar to **momentum**!) and $\vec{\hat{v}}_t$ to scale the learning rate (similar to **RMSprop**!).

$$\vec{w}_{t+1} \leftarrow \vec{w}_t - \frac{\eta}{\sqrt{\vec{\hat{v}}_t + \varepsilon}} \vec{\hat{m}}_t$$

  We have arrived at the Adam update rule!

- To use in Keras, set `optimizer='adam'`.

AdaMax (*Kingma and Ba, 2014*)

- Adam has used the L_2 norm of the gradient updates to compute $\vec{v}_t$—but using the L_p norm is also possible:

$$\vec{v}_t = \beta_2^p \vec{v}_{t-1} + (1 - \beta_2^p)|\vec{g}_t|^p \qquad \vec{u}_t = (\vec{v}_t)^{1/p}$$

- While values of $p > 2$ are generally unstable, stable behaviour re-emerges at L_∞, yielding the simple *AdaMax* algorithm:

$$\vec{u}_0 = \vec{0} \qquad \vec{u}_t = \max(\beta_2 \cdot \vec{u}_{t-1}, |g_t|) \qquad \vec{w}_{t+1} \leftarrow \vec{w}_t - \frac{\eta}{\vec{u}_t}\hat{\vec{m}}_t$$

- The max operator does not bias towards zero, and therefore there is no need to compute a bias-corrected $\vec{u}_t$.
- To use in Keras, set `optimizer='adamax'`.

Nadam (*Dozat, 2016*)

- As we noted, Adam leverages the ideas from both *momentum* and *RMSprop*—how about adding *Nesterov momentum*?
- To simplify, we modify the steps of Nesterov momentum—rather than using $\vec{w}_t - \beta_1 \vec{\hat{m}}_{t-1}$ as a position to compute gradients, we compute them at $\vec{w}_t$ (as with ordinary momentum), but take an additional step in the direction of $\beta_2 \vec{\hat{m}}_t$ at the end!
- Integrating this into Adam's update rule…

Nadam (*Dozat, 2016*)

- Expanding Adam's update rule, we obtain:

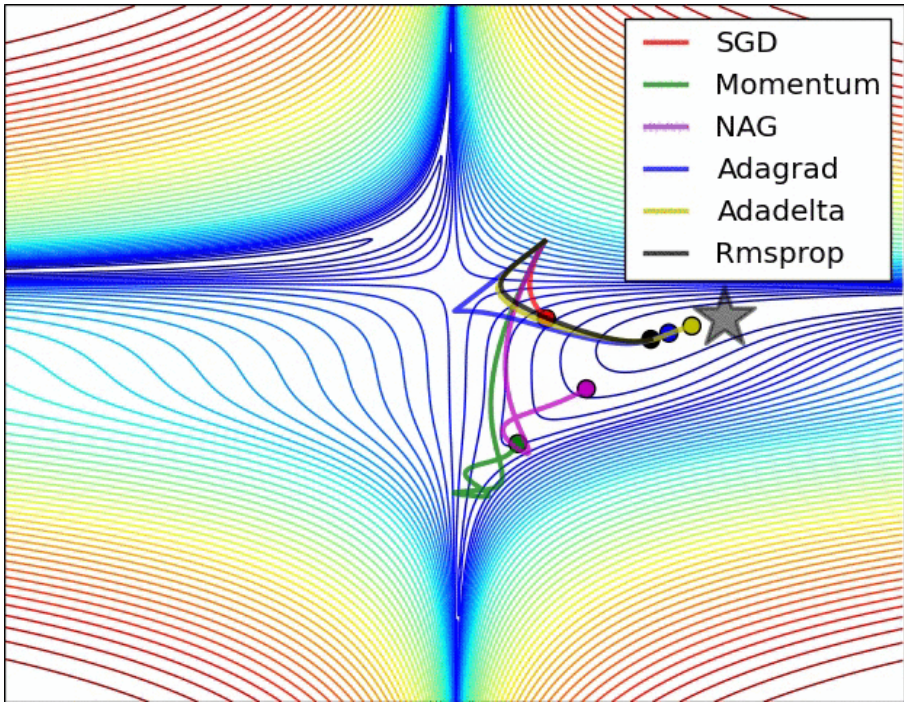
$$\vec{w}_{t+1} \leftarrow \vec{w}_t - \frac{\eta}{\sqrt{\hat{\vec{v}}_t + \varepsilon}} \left(\frac{\beta_1 \vec{m}_{t-1}}{1 - \beta_1^t} + \frac{(1 - \beta_1)\vec{g}_t}{1 - \beta_1^t} \right)$$

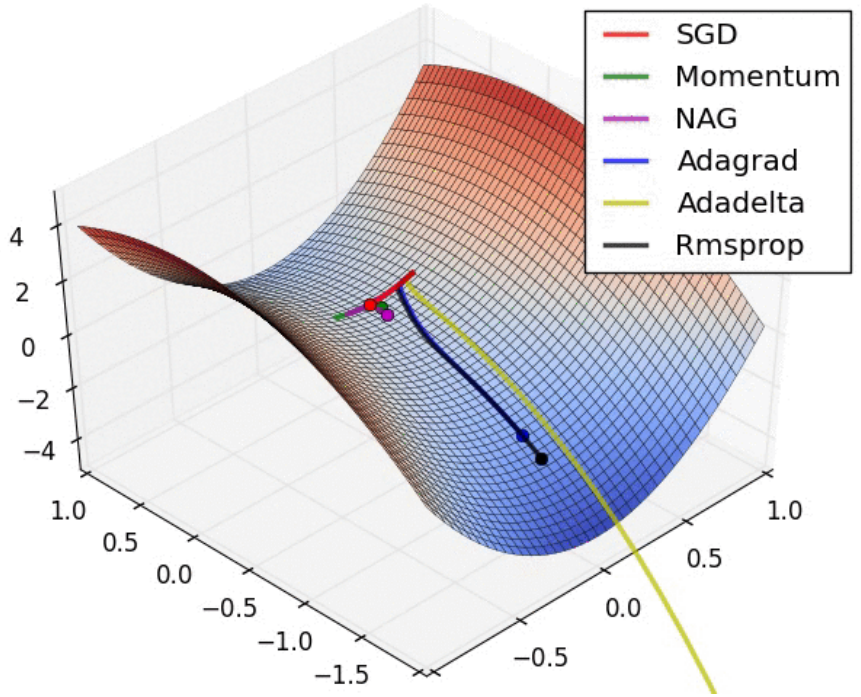
$$= \vec{w}_t - \frac{\eta}{\sqrt{\hat{\vec{v}}_t + \varepsilon}} \left(\beta_1 \hat{\vec{m}}_{t-1} + \frac{(1 - \beta_1)\vec{g}_t}{1 - \beta_1^t} \right)$$

- Now, just replace $\hat{\vec{m}}_{t-1}$ with $\hat{\vec{m}}_t$ (to do the "look-ahead" step as per the previous slide) to obtain the **Nadam** update rule!

$$\vec{w}_{t+1} \leftarrow \vec{w}_t - \frac{\eta}{\sqrt{\hat{\vec{v}}_t + \varepsilon}} \left(\beta_1 \hat{\vec{m}}_t + \frac{(1 - \beta_1)\vec{g}_t}{1 - \beta_1^t} \right)$$

Visualisation (credit: *Alec Radford*)



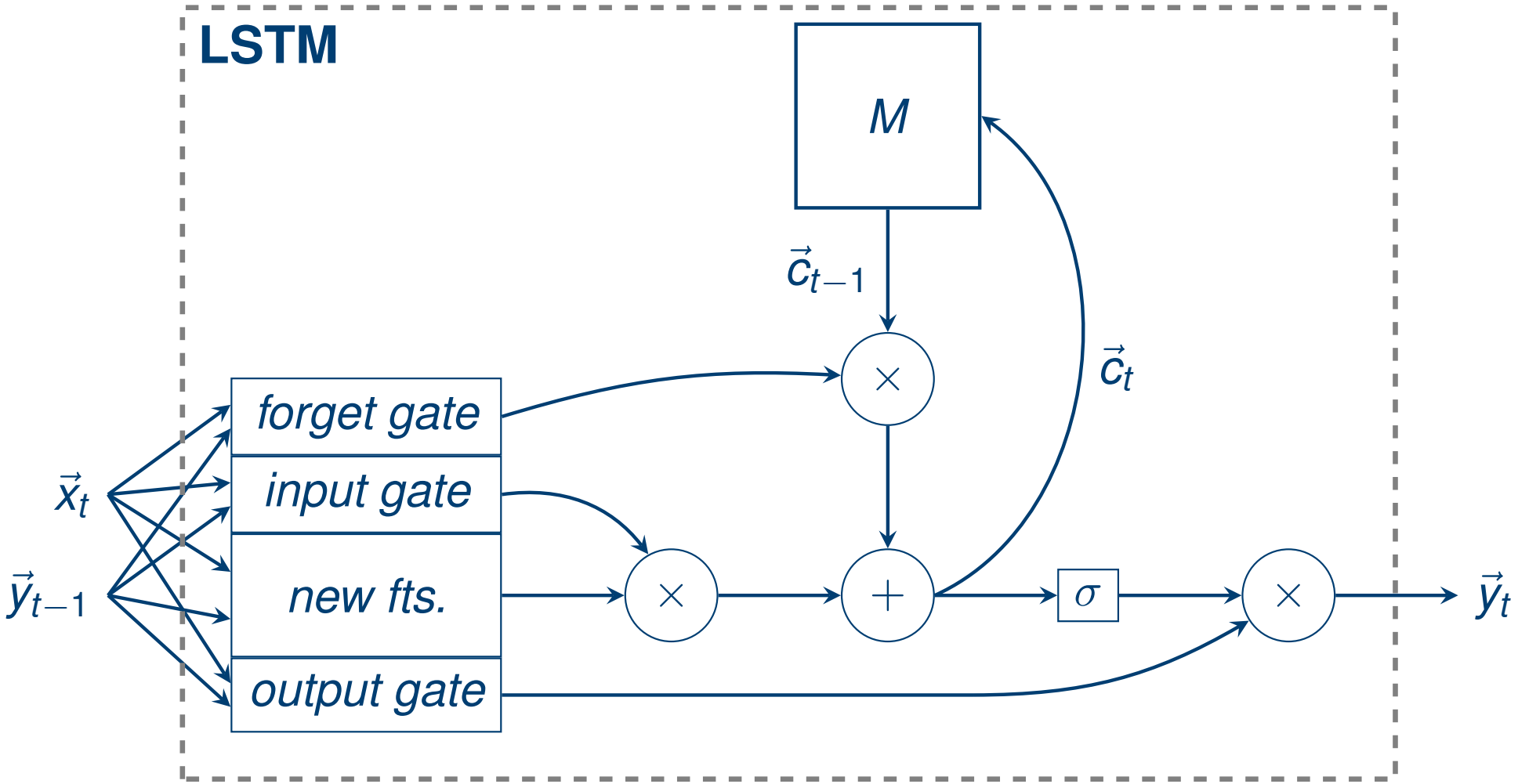


Adaptive methods generally outperform SGD—especially when it comes to escaping saddle points.

Being nice to RNNs

- *Recurrent neural networks* are somewhat special:
  - They have extensive *weight sharing* through time;
  - They often have to cope with *variable-length* sequences;
  - They may not have access to large batch sizes of data;
  - . . .
- This introduces a specific set of tricks for optimising them. . .
- I will focus on the *long short-term memory* (LSTM) model here; but many of these ideas will work on other cell types as well.

Recall: LSTM computational graph



Recall: LSTM equations

$$
\left.
\begin{aligned}
\vec{i}_t &= \mathit{logistic}\left(\mathbf{W}_i \vec{x}_t + \mathbf{U}_i \vec{y}_{t-1} + \vec{b}_i\right) \\
\vec{f}_t &= \mathit{logistic}\left(\mathbf{W}_f \vec{x}_t + \mathbf{U}_f \vec{y}_{t-1} + \vec{b}_f\right) \\
\vec{o}_t &= \mathit{logistic}\left(\mathbf{W}_o \vec{x}_t + \mathbf{U}_o \vec{y}_{t-1} + \vec{b}_o\right)
\end{aligned}
\right\} \mathit{gates}
$$

$$
\left.
\vec{ft}_t = \tanh\left(\mathbf{W}_{ft} \vec{x}_t + \mathbf{U}_{ft} \vec{y}_{t-1} + \vec{b}_{ft}\right)
\right\} \mathit{new\ features}
$$

$$
\left.
\vec{c}_t = \vec{ft}_t \otimes \vec{i}_t + \vec{c}_{t-1} \otimes \vec{f}_t
\right\} \mathit{update\ cell}
$$

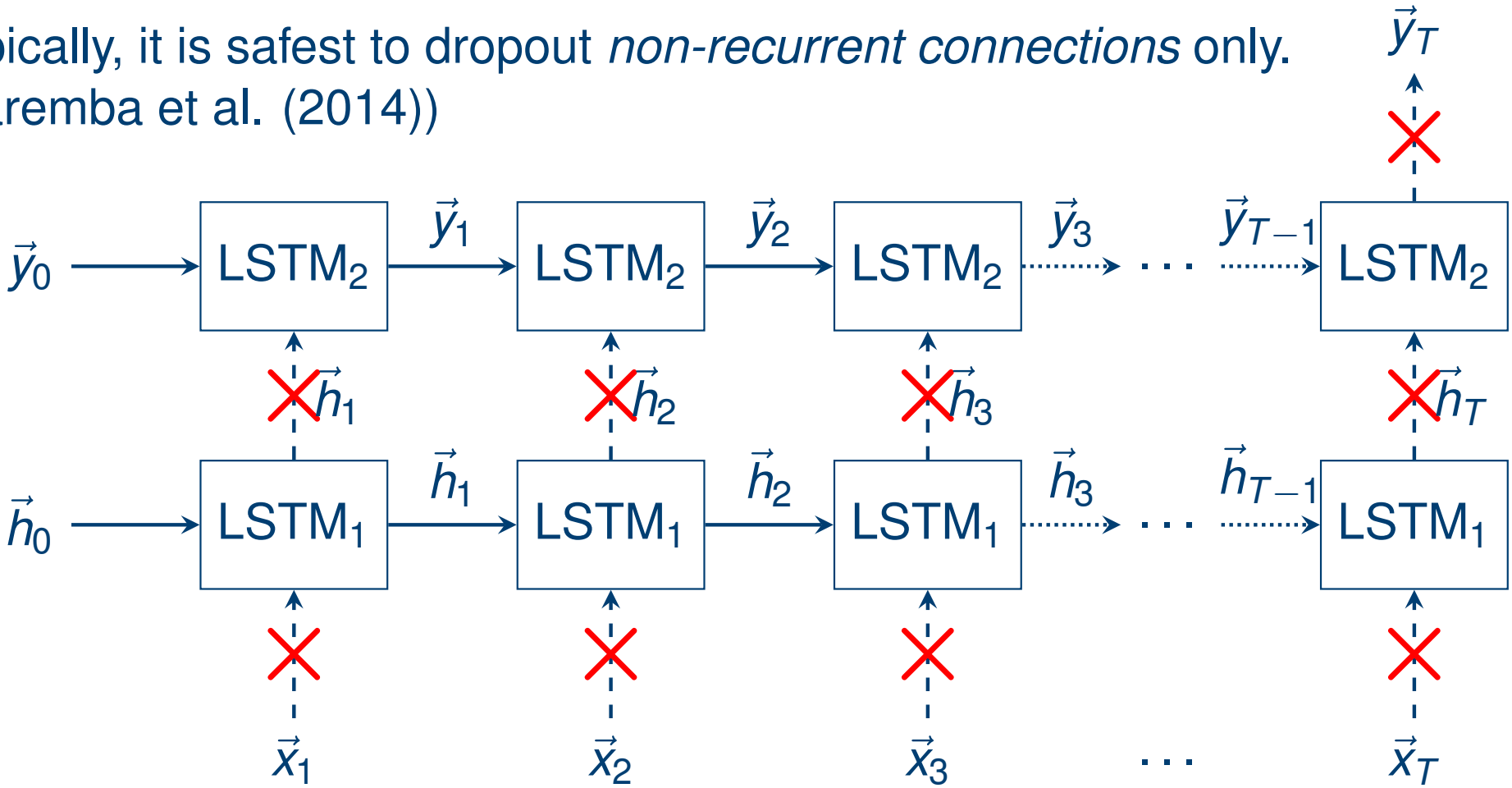
$$
\left.
\vec{y}_t = \tanh\left(\vec{c}_t\right) \otimes \vec{o}_t
\right\} \mathit{output}
$$

LSTM initialisation (*Jozefowicz et al., 2015*)

- It is important to choose the initial parameter values to help the LSTM learn effectively in the early stages!
- Sensible initialisations are (Keras does these *automatically*):
  - $\mathbf{U}_*$ with *orthogonal* initialisation (help combat vanishing gradients even further; eigenvalues ~ 1).
  - $\vec{b}_f = \vec{1}$ (to encourage long-term dependencies early on);
  - *Xavier initialisation* (Glorot and Bengio (2010)) for all other weights (*recommended* for sigmoid activations).
- For RNNs, the *Adam* (Kingma and Ba (2014)) and *RMSProp* (Tieleman and Hinton (2012)) optimisation algorithms work particularly well.

Dropout in LSTMs

Typically, it is safest to dropout *non-recurrent connections* only. (Zaremba et al. (2014))



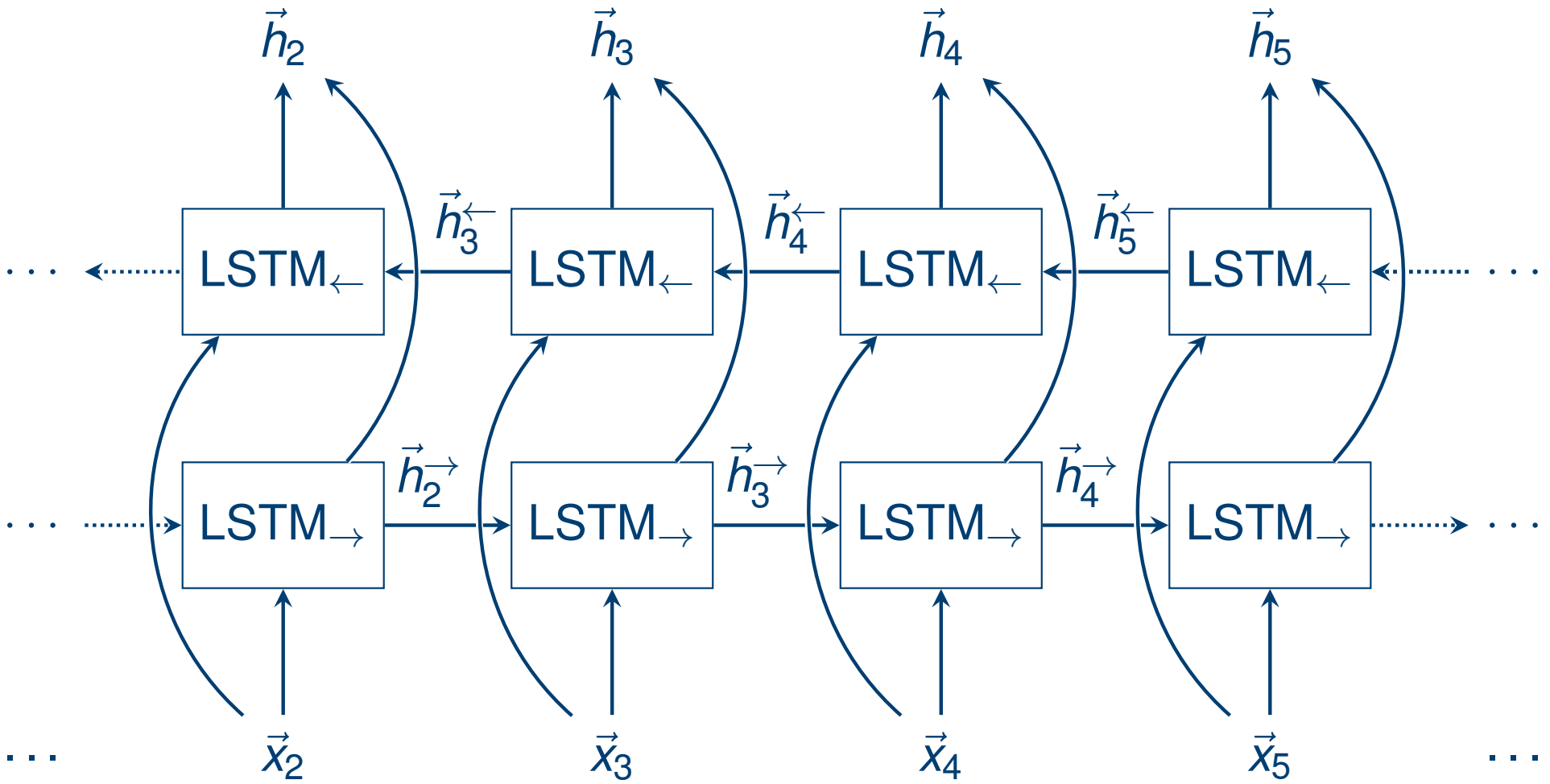
Normalisation techniques

- Generally, batch normalisation may be less applicable, if sequences will have a wide variety of lengths (as many entries will end up missing, leading to weak approximations of the step statistics).
- Techniques such as *layer normalisation* and *weight normalisation* seem to rectify this situation well (albeit while being a bit harder to control).
- **Recurrent batch normalisation** (*Cooijmans et al., 2017*) batch-normalises the hidden-to-hidden connections of an RNN, achieving solid results.

Bidirectional LSTM

- Very often, the dependencies within sequential data are not just in *one* direction, but may be observed in *both*!
- Examples: DNA strands, words in a sentence, ...
- Bidirectional layers exploit this by combining the features obtained going in both directions simultaneously.
- Very simple to deploy in Keras (`Bidirectional` wrapper around recurrent layers).

Bidirectional LSTM in action



Thank you!

Questions?

petar.velickovic@cst.cam.ac.uk

Special thanks:
César Laurent (*Montréal Institute for Learning Algorithms*)

Reading material:
‘Deep Learning’, Chapter 7 (Regularisation)
‘Deep Learning’, Chapter 8
(Initialisation, Optimisation, Normalisation)
‘Deep Learning’, Chapter 10 (RNN pointers)